Modeling Computer *In*security

Sophie Engle, Sean Whalen and Matt Bishop
Department of Computer Science
University of California, Davis
{*engle, whalen, bishop*}*@cs.ucdavis.edu*

Abstract

In this paper, we present a formal model of computer security based on the universal Turing machine. This model allows us to conclude that given a machine and policy, the problem of determining if that machine is secure is not recursively enumerable. However, two related problems are solvable. The inverse problem of determining if a machine is insecure is recognizable. Additionally, determining if the current configuration of a machine is secure is decidable. Given these theoretical results, we propose a shift in how we discuss "security" in practice.

1. Introduction

Computers are becoming ever more embedded in everyday life. Computers run our banks, provide us directions in our cars, tell us when we are out of milk in our fridges, provide us restaurant reviews on our cell phones, archive our personal information, and even assist us in electing our government representatives. As the ubiquity of computers increase, so does the importance of computer security. The presence of a vulnerability could cause a minor annoyance in the form of lost time, cost billions in information theft, bring down global networks, or even throw an election.

However, while a large volume of work has been done on preventing and defending against known vulnerabilities, it is not clear where vulnerabilities fit into the classic formal model of modern computers - the universal Turing machine. This work attempts to formally answer two fundamental questions: what is a vulnerability, and when is a machine secure? We do this by providing a formal model for computer security which uses universal Turing machines to model computer systems.

By formally defining these concepts, we gain a precise method for discussing and analyzing security, policy, and vulnerabilities. As part of this formalization, we have been able to precisely define the notion of *conditions* to describe both vulnerabilities and security policy. Using our model as a foundation, this may lead to a more precise method for characteristic-based classification of vulnerabilities [4].

Finally, we discover that even with a specific policy, the problem of determining if a machine is secure is not recursively enumerable. We then propose a shift in focus in vulnerability research. Instead of focusing on how to evaluate the security of a machine (which we find to be theoretically impossible), we should focus on evaluating the *insecurity* of a machine or on the *current* security of a machine.

2. Background

We use *universal Turing machines* as a theoretical model of modern computer systems. Informally, a Turing machine is a *state machine* with an infinite tape, capable of reading and writing symbols, moving left or right along the tape as instructed.

Formally defined, a **Turing machine** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_\mathrm{a}, q_\mathrm{r})$ where Q is the set of states, Σ is the input alphabet, Γ is the output/tape alphabet where $\Sigma \subseteq \Gamma$, $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function where $\{\mathrm{L}, \mathrm{R}\}$ indicate the head is to move left or right respectively, $q_0 \in Q$ is the start state, $q_\mathrm{a} \in Q$ is the accept state, and $q_\mathrm{r} \in Q$ is the reject state where $q_\mathrm{a} \neq q_\mathrm{r}$ [2].

A **universal Turing machine** is a Turing machine which is capable of simulating any other Turing machine [2]. We specifically use *decidable* universal Turing machines as our system model. A **decidable Turing machine** is one that always accepts or rejects, and as a result always halts [2].

We also occasionally refer to a **multi-tape Turing machine** which is a Turing machine with multiple infinite tapes [2]. Multi-tape Turing machines are computationally equivalent to normal Turing machines.

A **configuration** of a Turing machine captures the current state, the content of the tape, and the current location of the head [2]. The configuration is often captured as the string uqv where q is the current state, the string uv gives the contents of the tape, and the first position of v gives the current location of the head. If the state of a configuration is the state q_a, it is considered an **accepting configuration**.

3. Vulnerabilities, Part 1

Computers are highly complex and inevitably have numerous bugs due to human error. However, only a subset of these bugs are considered *vulnerabilities* and hence a threat to security. As part of our formal model, we attempt to capture the fundamental and intuitive difference between bugs versus vulnerabilities.

To identify this difference, we must first explore what makes a computer *secure* which is in turn dependent on its *security policy*. As such, we start by exploring how to formally define security policy for a deterministic universal Turing machine.

4. Security Policy

A *security policy* defines what it means for a specific machine to be "secure" by specifying what is or is not allowed. In practice, security policies may be implicit, informally defined, or explicitly defined using rigorous mathematical notation.

In this section, we explore the formal definition of security policy, and why one common definition is insufficient for our model.

4.1. Policy as a Partition

One definition of security policy is, "a statement that partitions the states of the system into a set of *authorized*, or *secure*, states and a set of *unauthorized*, or *nonsecure*, states." In this sense, a system is *secure* if it "starts in an authorized state and is unable to enter an unauthorized state" [1]. However, we argue that a simple partition to describe security policy is insufficient.

Assume that we have a security policy defined as a partition of authorized and unauthorized states in a machine. If it is possible for the machine to be secure, then it is impossible to reach an unauthorized state from the start state. In this case, these unauthorized states may be considered *useless states* and removed from the system.

If an unauthorized state is reachable from the start state, then the machine is not secure. In this case, removing the unauthorized state may impede operations necessary in the machine. Either the machine is poorly designed, or (more likely) these unauthorized states are necessary under certain circumstances and may not be removed from the machine. For example, state q_i may be necessary for a root user to perform maintenance operations, but should be unauthorized under all other circumstances.

Policy as a partition is unable to capture this. Policy should capture more than what is authorized or unauthorized, but also *when* it is authorized or unauthorized. For this, we need a more robust definition of policy.

4.2. Policy as a Language

Instead of defining policy by a set of authorized *states*, we focus on defining policy by a set of authorized *configurations*. Note that *accepting* configurations (defined earlier) are different from *authorized* configurations, which may include any state.

To capture the conditional nature of policy, we define the concept of a **policy condition** as a set (or language) of authorized configurations. For example, consider:

$$\underbrace{\texttt{user}:\texttt{root}\,\Gamma^*}_{u}\; q_i\; \underbrace{\Gamma^*}_{v}$$

This regular language (where Γ^* represents any string of tape characters) is a policy condition indicating that state q_i is authorized when the tape begins with `user : root`.

We define **security policy** as the union of policy conditions. The result is a language of authorized configurations for a Turing machine. By using policy conditions as a basis for our language, we capture not only which states are authorized, but also when they are authorized.

We define two special cases of policy conditions. A **state policy condition** (or state condition for short) is a policy condition which does not depend on the tape. For example, consider the regular expression $\Gamma^*\, q_j\, \Gamma^*$. This state policy condition indicates that the state q_j is authorized no matter what is on the tape. The definition of state policy conditions leads us to our first theorem:

> **Theorem 4.1:** A security policy defined exclusively by state policy conditions is equivalent to a security policy defined as a partition of authorized and unauthorized states.
>
> PROOF OUTLINE: Suppose we have a security policy defined as a partition of authorized and unauthorized states. We are able to create a language of authorized configurations by creating a state policy condition for each authorized state and taking the union all of these conditions. In the other direction, if we have a language of state policy conditions we can union together all of the states included in the state policy conditions to create the set of authorized states.

We also define a **tape policy condition** (or tape condition for short) as a policy condition which does not depend on the state or on the position of the read head. For example, suppose Q is the set of all states and we have:

$$Q\,abc \cup a\,Q\,bc \cup ab\,Q\,c \cup abc\,Q$$

This tape policy condition indicates that any state is authorized when abc is the current tape. As shorthand, we indicate tape conditions as just the tape contents with no reference to any states. In this example, we'd just state the tape

condition is abc. Tape conditions may be useful when defining policies such as the necessary requirements for a strong password on a machine (must be a certain length, must have digits and characters, and so forth).

As our definition of policy allows for more than just state policy conditions, we are able to state a corollary to our previous theorem:

> **Corollary 4.2:** Security policy defined as a language is more expressive than security policy defined as a partition of states.

The first example policy condition is proof of this. We are able to capture that q_i is authorized only under a specific condition. Using a partition of states, q_i must be listed as either authorized or unauthorized no matter the tape contents.

4.3. Policy Representation

Defining security policy as a language is not the same as the notion of "policy languages." A *policy language* is defined as "a language for representing a security policy" which may be high-level or low-level in nature [1]. Policy languages in this sense attempt to capture some abstract notion of policy in a way understandable to modern programs and systems. Consequently, policy and policy languages are two separate notions. Our definition unifies the notion of policy with the representation of policy.

There are, however, many different classes of languages which may be used for security policy. Our examples so far have used *regular expressions* to specify a security policy, but policy does not have to be limited to the class of regular languages. For example, we may instead try to use the class of *recursively enumerable languages* for policy, which may be recognized by Turing machines.

There are two major restrictions to which classes of languages may be used for security policy. The first restriction comes from the theoretical limits of computation. Turing machines are the most powerful model of computation, and there exist some languages which may not be captured by Turing machines [2]. As a result, the class of recursively enumerable languages represent the most powerful class of languages we are able to compute and hence use for security policy.

The second of these restrictions comes from the nature of security policy. In practice, security policy is often defined by what is unauthorized versus authorized. For example, suppose state q_k is unauthorized when the tape begins with `user : xander`. The unauthorized policy condition may be described as:

$$\texttt{user : xander}\,\Gamma^*\,q_k\,\Gamma^*$$

From this, we can derive the language of authorized configurations by using complementation:

$$\overline{\texttt{user : xander}\,\Gamma^*\,q_k\,\Gamma^*}$$

Security policy is defined as a language of authorized configurations, but we must also be able to derive what configurations are unauthorized to test for security. Again, this is accomplished using complementation. As such, we must be able to recognize both the language of authorized configurations and the language of unauthorized configurations. This is only possible with languages that are closed under complementation, which recursively enumerable languages are not [2]. Therefore, we only consider *recursive* or *decidable* languages, which may be decided by Turing machines [2].

Both of these restrictions illustrate the difference between what is *ideal* in specifying policy versus what is *feasible*. While we may be able to express an ideal policy using natural language, this representation is infeasible to implement at a machine level. To address the disconnect between ideal and feasible policy, we turn to the *Unifying Policy Hierarchy Model* as described in the next section.

4.4. Policy Hierarchy

As a result of technological limitations and configuration complexity, the policy implemented on a particular machine may not match the intentions of the policy administrator. This distinction between *implemented* versus *intended* is critical when discussing security policy. To capture this distinction, we adapt the *Unifying Policy Hierarchy Model* which provides a hierarchical classification of four different types of policy [3]. Each policy maps a query for a subject attempting to perform an action on an object in the system onto a valid or invalid response.

At the highest level of the hierarchy, the **Oracle Policy** defines an abstract oracle which decides if a given subject is allowed to execute a specific action on an object in the system. The oracle can answer queries about subjects, objects, and actions existing outside the system, whereas lower level policies are limited to internal entities. The role of the oracle is approximated in reality by a system administrator.

For example, a password-based system cannot distinguish between two people accessing the same account from the same keyboard with the same password. In reality they are different people, but the system cannot prevent the person Xander from typing Yasmin's username and password if he obtains them. The oracle, however, exists outside the system and can decide that the person Xander cannot login as the account `Yasmin`. Thus, there are intended policy decisions which cannot be implemented on an actual system and can only be decided by an oracle.

The policy which may be practically implemented on the system is the **Feasible Policy**. The policy decides if a subject, object, and action are allowed under the restrictions imposed by working within the system itself.

The policy intended for a system is the **Machine Policy**. In the space of all feasible policies for a system, a particular policy is configured by an administrator.

Finally, the **Real-Time Policy** of a system is the policy which is currently active. The implementable, intended policy for the system may differ from the policy actually in place due to policy violations.

4.5. Policy Violations

Using the Unifying Policy Hierarchy Model, we are able to define different types of policy violations and narrow the types of violations we consider in our security research.

The most abstract policy violation is the **inherent policy violation**, which occurs whenever oracle policy does not match the feasible policy. These violations are often the result of technological limitations on what portions of the oracle policy may be expressed, or due to poor or ambiguous expression of the oracle policy.

Whenever the feasible policy disagrees with the machine policy, we say a **configuration policy violation** has occurred. These violations are often the result of misconfiguration when the feasible policy was applied on the machine, often due to the complexity of this process or policy, or poor interface design.

Finally, a *real-time policy violation* occurs whenever the machine policy disagrees with the real-time policy. In relation to our model, we consider machine policy to be the security policy, and the current configuration to be the real-time policy of the machine. Therefore, we can state that a **real-time policy violation** occurs whenever the current configuration of a Turing machine is unauthorized as specified by its security policy.

We are only focused on when the machine behaves in a way which violates its policy, and hence focus solely on real-time policy violations. Addressing the limitations of technology which cause inherent policy violations is an area for future work, as is addressing how to avoid configuration policy violations.

5. Vulnerabilities, Part 2

With the definition of security policy and of real-time policy violations, we are now able to determine *when* a vulnerability occurs. However, our definition of vulnerability is not focused solely on the policy violation, but also on the conditions which lead to the violation.

Computation in a Turing machine is defined as a series of configurations, sometimes called the **computation history**. For example if we have the series of configurations:

$$\texttt{AB}\, q_1\, \texttt{cde} \rightarrow \texttt{ABC}\, q_2\, \texttt{de}$$

We are able to determine that the Turing machine followed a transition to state q_2 which read a `c`, replaced it with a `C`, moved the tape head right one position.

Using this computation history, we can identify not only what configuration violated policy, but those configurations which led to that policy violation. Again, we define a notion of **condition** to be a language of configurations. We use conditions to describe the computation history leading to a policy violation. For example, suppose a buffer overflow lead to a policy violation. If the buffer is of size n, we could specify the tape condition Γ^{n+1} to describe what caused the violation.

Using the notions of real-time policy violations and conditions, we can finally define a *vulnerability*:

> **Definition 5.3:** We define a **vulnerability** as the pair (V, C), where V is the unauthorized configuration causing a real-time policy violation, and C is a set of conditions describing the computation history leading to V.

The result is a vulnerability definition which not only captures the unauthorized configuration, but also the conditions which must be satisfied for that configuration to occur.

6. Computer Security

Intuitively, a machine is secure when it has no vulnerabilities. If there are no vulnerabilities in the machine, then there are no policy violations. This is only true when the machine is unable to enter an unauthorized configuration. Following this line of reasoning, we state a machine is **secure** if and only if it is unable to enter an unauthorized configuration as specified by its security policy.

We define the language ***SECURE*** to be set of all pairs $\langle M, P \rangle$ where M and P are decidable Turing machines and M never enters a configuration not in $L(P)$. We reduce the problem of determining if a machine is secure to determining if a machine and policy belong to this language.

> **Theorem 6.4:** The language *SECURE* is not recursively enumerable.
>
> INTUITION: The intuition behind this is based on the fact that there are potentially an infinite number of possible configurations for any Turing machine. To be positive the machine is unable to enter an unauthorized configuration, each of these configurations

would need to be tested. Even with a decidable policy language and decidable Turing machine, this test would never halt.

PROOF OUTLINE: The idea behind a more formal proof is to use mapping reducibility and reduce the language E_{TM} to *SECURE*. The language E_{TM} includes all Turing machines M where $L(M) = \emptyset$, and is known to be unrecognizable [2]. Given an input M, build an enumerator P which simulates M on an input string x and outputs every configuration entered by M *except* any accepting configurations[1]. If the language of M is empty, then there are no accepting configurations and M will only enter those configurations specified by $L(P)$. Otherwise, an accepting configuration is reachable in M and M will enter a configuration not authorized by $L(P)$. Therefore, $\langle M \rangle \in E_{\mathrm{TM}} \Leftrightarrow \langle M, P \rangle \in$ *SECURE*.

This allows us to make a more general statement about security:

Corollary 6.5: The general problem of security, even with a specific security policy, is unsolvable.

This result matches other formal results (as discussed in related work), and the intuition of security professionals today. In fact, most measures of "security" are actually measures of *insecurity*, which we find to be a solvable problem.

7. Computer Insecurity

Insecurity is the complement of security. A machine is **insecure** if it is capable of entering an unauthorized configuration according to the security policy. We define the language ***INSECURE*** as the complement of *SECURE* – or simply $\overline{\textit{SECURE}}$. While *SECURE* is not recursively enumerable, we find that is not the case with the language *INSECURE*.

Theorem 7.6: The language *INSECURE* is recursively enumerable.

PROOF OUTLINE: Consider a Turing machine M' which on input $\langle M, P \rangle$ nondeterministically simulates M on a string w. If M ever enters a configuration not in the language of P, then it should reject. Since we only consider decidable Turing machines for M and P, this check will always halt. However, if there does not exist such a w, then our nondeterministic Turing machine M' will never halt. As such the language *INSECURE* is recursively enumerable, but not recursive (or decidable).

[1]Recall from our background discussion that we are only considering decidable Turing machines as our system model, which allows us to build this enumerator.

This result shows that attempting to determine if a machine is insecure is "easier" than attempting to determine if it is "secure." This result matches the intuition of security professionals today. Professionals today focus on determining if a machine is insecure by checking for known vulnerabilities or insecure programming practices. Additionally, machines which haven't yet been marked insecure are continually checked against new vulnerabilities.

This suggests that the focus of security research is and should be on the insecurity of a machine, not on the security of a machine. While this may be arguing the semantics of security, it has a significant impact on the public perception. Claiming a machine is "secure" is different from "not known to be insecure," and the latter helps stress the important of constantly checking for new vulnerabilities.

From these results, we know the general problem of determining if a machine is secure is unsolvable and that determining if a machine is insecure is only recursively enumerable. Instead, a more focused approach on special cases of these problems is required to make security fully solvable. One type of special-case security is *real-time security*, as we discuss in the next section.

8. Real-Time Security

While determining if a machine is secure with respect to its policy is an undecidable problem, we are able to decide if the current configuration of a machine is secure. We call this concept *real-time security*. Specifically, we say a machine is currently **real-time secure** if and only if every configuration in its computation history is authorized by the security policy. This leads us to the following claim:

Theorem 8.7: Real-time security is decidable.

PROOF OUTLINE: This is only true because we limit ourselves to decidable (or recursive) policies. As the machine moves from one configuration to another during computation, we are able to decide if that configuration belongs to our language of authorized configurations as given by our security policy. If the machine ever enters an unauthorized configuration, a special flag can be set on the tape.

While this does not tell is if the machine will be secure in the future, it does at least allow us to determine if it is currently secure. This allows us to react when the machine

becomes insecure, but does not allow us to predict or defend against the potential vulnerabilities.

Unfortunately, the concept of real-time security has limited application in practice. Aside from the fact that real-world systems and polices are too complex to describe as Turing machines, constantly checking if the machine is still secure is likely prohibitively time consuming. However, real-time security is at least achievable theoretically, unlike the general notion of machine security. This validates the generally accepted assumption that detecting vulnerabilities is an easier problem than preventing them.

9. Example Machine and Policy

Since our definitions depend on configurations, we must fully specify the Turing machine and policy to determine if a machine is insecure or real-time secure. While it is possible to provide a Turing machine and security policy illustrating vulnerabilities such as buffer overflows, the complexity of the specification overwhelms the example itself. We have instead chosen to provide a simple example Turing machine and security policy to illustrate the concepts of insecurity and real-time security. This example illustrates the use of configurations in determining these properties without burying the reader in the details of a more complicated Turing machine specification.

Suppose we have a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_\mathrm{a}, q_\mathrm{r})$ with input alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, tape alphabet $\Gamma = \Sigma \cup \{\textvisiblespace\}$ where $\textvisiblespace$ indicates a blank space on the tape, and the states $Q = \{q_0, q_\mathrm{a}, q_\mathrm{r}\}$. Let M take any digit n on the tape and replace it with $n + 1 \bmod 10$. This gives the transition function:

$$\delta(q_0, x) = \begin{cases} (q_0, x + 1 \bmod 10, \mathrm{R}) \text{ if } x \in \Sigma \\ (q_\mathrm{a}, x, \mathrm{R}) \text{ otherwise} \end{cases}$$

For example, given input 3579 the computation of our Turing machine M would complete with 4680 on the tape as show in the following computation history:

$$\begin{aligned} q_0\,3\,5\,7\,9\,\textvisiblespace &\rightarrow 4\,q_0\;5\;7\;9\;\textvisiblespace \\ &\rightarrow 4\;6\;q_0\;7\;9\;\textvisiblespace \\ &\rightarrow 4\;6\;8\;q_0\;9\;\textvisiblespace \\ &\rightarrow 4\;6\;8\;0\;q_0\;\textvisiblespace \\ &\rightarrow 4\;6\;8\;0\;\textvisiblespace\;q_\mathrm{a} \end{aligned}$$

Our security policy could be that we never want a Fibonacci number to appear on the tape. The language of Fibonacci numbers is a decidable language, and hence there exists a Turing machine F which decides this language. Therefore our policy P would be the language of *tape conditions* $P = \overline{L(F)}$.

We know from section 8 that we can determine the real-time security of a machine. For example, the only numbers that appear on the tape given input 3579 are:

$$3579, 4579, 4679, 4689, 4680$$

Since none of these are Fibonacci numbers, we can say for input 3579, the machine M is always real-time secure with respect to P.

However, is this system secure? We know from section 6 that we are unable to say M is secure. However, section 7 tells us we may be able to determine if M is insecure. Consider the input 1484. The Turing machine will compute as follows:

$$\begin{aligned} q_0\,1\,4\,8\,4\,\textvisiblespace &\rightarrow 2\,q_0\;4\;8\,4\,\textvisiblespace \\ &\rightarrow 2\;5\;q_0\,8\,4\,\textvisiblespace \end{aligned}$$

Notice here that number 2584 appears on the tape, and is a Fibonacci number (with $n = 18$). At this point, we can say that M is not currently real-time secure, and furthermore that M is insecure with respect to its security policy.

10. Related Work

Other models of vulnerabilities generally assume an unstated policy or model flaws generally accepted to be vulnerabilities, for example buffer overflows. For example, Chen et al. [5] use a finite state machine model to reason about vulnerabilities identified in the bugtraq database, a standard repository of information about vulnerabilities. They then use this model to reason about other (potential) vulnerabilities. Throughout, the authors assume that buffer overflows are vulnerabilities. This is true when the overflow allows a user to add privileges which that user is not authorized to have. But buffer overflows in unprivileged programs do not do this, and so under most policies are not vulnerabilities.

Our work is more general, in that it takes policy into account. Further, it provides a mechanism to define explicit conditions that compose the vulnerability.

Much work has applied modeling to specific systems and situations in order to analyze vulnerabilities. Shahriari and Jalili [6] use a variant of the Take-Grant Protection Model to analyze vulnerabilities in networks. Zakeri et al. [7] have used description logic to model the TCP/IP protocol to find vulnerabilities. Frantzen et al. have applied dataflow models to analyze vulnerabilities in firewalls. This work points out the value of applying formal modeling to systems. Our work speaks more to the characterization of what vulnerabilities are and how they interact with different layers of policies, and is not tied to any particular system.

The use of formal models to analyze attacks and techniques for attacks relates to our work in that the attacks set up conditions needed to exploit vulnerabilities. Clearly, the conditions that must hold for attacks to succeed are

those that create one or more vulnerabilities. Templeton and Levitt captured this notion in their requires/provides model [9]. Even though their model is informal, a formal model analogous to ours can be readily constructed. At a higher level, Jha et al. [10] treat vulnerabilities as aspects of safety properties that can be violated, and uses that to develop attack graphs automatically. Again, our work is more foundational, and focuses on the definitions of what a vulnerability is, and how to model it.

11. Conclusion

In practice, we do not have infinite time and computers do not have infinite memory. Problems which are solvable by Turing machines may take too much space or too much time to solve on modern systems. Furthermore, modern systems and security policies are too complex to specify as Turing machines.

However, theoretical work does place an upper bound on what is achievable in computer security and suggests directions to focus research. Our theoretical results suggest measures of insecurity or real-time security may be more achievable than measures of security.

Specifically, we show that it is theoretically impossible to determine if a generic machine is secure with respect to its security policy. The complement, computer *in*security, is a problem we can recognize. We argue that any machine claimed to be secure is in fact not known to be insecure.

We can avoid known insecurities, but we must never grow complacent. A system which is not known to be insecure today may become insecure tomorrow. Systems must be continually checked for vulnerabilities or insecure coding practices. Emphasizing this is the notion of real-time security, which we find to be a decidable problem. We are able to determine if a machine is and has been secure, but not if it will always be secure. Once a vulnerability is detected, it may be addressed.

The "security" of voting systems is one example of where these results may be applied. Vendors make claims of security, but resist in-depth security code reviews. From these results, we know that such reviews are necessary to guarantee the system is not known to be insecure. This necessity does not come from poor programming or weak design, but from the very nature of security itself.

Finally, these results stress that methods for prevention of vulnerabilities must be based on empirical knowledge. We are only able to determine what makes a machine insecure today, and may only guess from that what problems we may face in the future.

While security versus insecurity may be an argument of semantics, it is important. It shapes public perception, and places emphasis on the evolving nature of the field. Much of computer security today involves studying insecurity by analyzing vulnerabilities and unsafe coding practices, or detecting and reacting to real-time or static vulnerabilities. Our work provides the theoretical foundation for intuition already held by security professionals today.

References

[1] Bishop, M., *Computer Security: Art and Science*, Addison-Wesley, Boston, 2003.

[2] Sipser, M., *Introduction to the Theory of Computation*, PWS Publishing Company, Boston, 1997.

[3] Carlson, A., "The Unifying Policy Hierarchy Model," Master's Thesis, University of California, Davis, 2006.

[4] Bishop, M., "Vulnerabilities Analysis," *Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection*, September, 1999, pp. 125–136.

[5] S. Chen, Z. Kalbarczyk, J. Xu, and R. Iyer, "A Data-Driven Finite State Machine Model For Analyzing Security Vulnerabilities," *Proceedings of the 2003 International Conerence on Dependable Systems and Networks*, June, 2003, pp. 605–614.

[6] H. Shahriari and R. Jalili, "Vulnerability Take Grant (VTG): An Efficient Approach To Analyze Network Vulnerabilities," *Computers and Security*, 26(5), August, 2007, pp. 349–360.

[7] R. Zakeri, R. Jalili, H. Shahriari, and H. Abolhassani, "Using Description Logics for Network Vulnerability Analysis," *Proceedings of the 2006 International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*, April, 2006, pp. 78–83.

[8] M. Frantzen, F. Kerschbaum, E. Schultz, and S. Fahmy, "A Framework for Understanding Vulnerabilities in Firewalls Using a Dataflow Model of Firewall Internals," *Computers and Security*, 20(3), May, 2001, pp. 263–270.

[9] S. Templeton and K. Levitt, "A Requires/Provides Model for Computer Attacks," *Proceedings of the 2000 New Security Paradigms Workshop*, September, 2000, pp. 31–38.

[10] S. Jha, O. Sheyner, and J. Wing, "Two Formal Analyses of Attack Graphs," *Proceedings of the 2002 Computer Security Foundations Workshop*, June, 2002, pp. 49–63.