An Attack Surface Metric

Pratyusa K. Manadhata, *Member*, *IEEE*, and Jeannette M. Wing, *Fellow*, *IEEE*

Abstract—Measurement of software security is a long-standing challenge to the research community. At the same time, practical security metrics and measurements are essential for secure software development. Hence, the need for metrics is more pressing now due to a growing demand for secure software. In this paper, we propose using a software system's *attack surface measurement* as an indicator of the system's security. We formalize the notion of a system's attack surface and introduce an *attack surface metric* to measure the attack surface in a systematic manner. Our measurement method is agnostic to a software system's implementation language and is applicable to systems of all sizes; we demonstrate our method by measuring the attack surfaces of small desktop applications and large enterprise systems implemented in `C` and `Java`. We conducted three exploratory empirical studies to validate our method. Software developers can mitigate their software's security risk by measuring and reducing their software's attack surfaces. Our attack surface reduction approach complements the software industry's traditional code quality improvement approach for security risk mitigation and is useful in multiple phases of the software development lifecycle. Our collaboration with SAP demonstrates the use of our metric in the software development process.

Index Terms—Code design, life cycle, product metrics, protection mechanisms, risk mitigation, software security.

1 Introduction

Measurement of security, both qualitatively and quantitatively, is a long-standing challenge to the research community and is of practical import to software industry today [1], [2], [3], [4]. The software industry is responding to a growing demand for secure software and is increasing its effort to create "more secure" products and services (e.g., Microsoft's Trustworthy Computing Initiative and SAP's Software LifeCycle Security efforts). Is industry's effort paying off? Are consumers getting more secure software and services? Security metrics and measurements could help both software developers to quantify improvements to their software's security and consumers to compare along a security dimension software alternatives with similar functionality.

In this paper, we introduce the notion of a software system's *attack surface* and present a systematic way to measure it. Intuitively, a system's attack surface is the set of ways in which an adversary can enter the system and potentially cause damage. Hence, the "smaller" the attack surface, the more secure the system.

1.1 Motivation

Software vendors traditionally focus on improving code quality to improve software security and quality; their effort aims toward reducing the number of design and coding errors in software. An error causes software to behave differently from the intended behavior defined by the software's specification; a vulnerability is an error that can be exploited by an attacker. In principle, we can use formal correctness proof techniques to identify and remove all errors in software with respect to a given specification and hence remove all of its vulnerabilities. In practice, however, building large and complex software devoid of errors, and hence security vulnerabilities, remains a difficult task. First, specifications, in particular explicit assumptions, can change over time, so something that was not an error can become an error later. Second, formal specifications are rarely written in practice. Third, formal verification tools used in practice to find and fix errors, including specific vulnerabilities such as buffer overruns, usually trade soundness for completeness or vice versa. Fourth, we do not know the vulnerabilities of the future, i.e., the errors present in software for which exploits will be developed in the future.

Software vendors have to embrace the hard fact that their software will ship with both known and future vulnerabilities and many of those vulnerabilities will be discovered and exploited. They can, however, reduce the risk associated with the exploitation; one way to do so is by reducing their software's attack surfaces. A smaller attack surface mitigates security risk by making the exploitation harder and by lowering the exploitation's damage. As shown in Fig. 1, the code quality effort and the attack surface reduction approach complement each other in mitigating security risk. Software developers can use attack surface measurements as a tool in multiple phases of software development to mitigate security risk, to prioritize testing effort, to choose a secure configuration, and to guide vulnerability patch implementation (Section 6.4).

1.2 Attack Surface Metric

We know from the past that many attacks on a system take place either by sending data from the system's operating environment into the system (e.g., buffer overflow exploitation) or by receiving data from the system (e.g., symlink attacks). In both of these types of attacks, an attacker connects to a system, using the system's *channels* (e.g.,

- *P.K. Manadhata is with Symantec Research Labs, 900 Corporate Pointe, Culver City, CA 90230. E-mail: manadhata@cmu.edu.*
- *J.M. Wing is with Carnegie Mellon University and the Computer and Information Science and Engineering Directorate at the US National Science Foundation, 4201 Wilson Boulevard, Suite 1105, Arlington, VA 22230. E-mail: wing@cs.cmu.edu.*

Manuscript received 23 Oct. 2009; revised 28 Apr. 2010; accepted 12 May 2010; published online 26 May 2010.
Recommended for acceptance by P. McDaniel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-10-0272.
Digital Object Identifier no. 10.1109/TSE.2010.60.

0098-5589/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

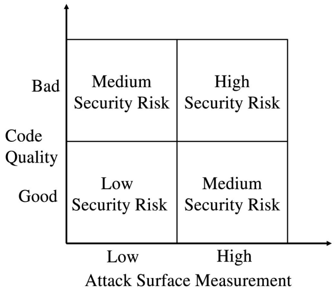

Fig. 1. Attack surface reduction and code quality improvement complement each other.

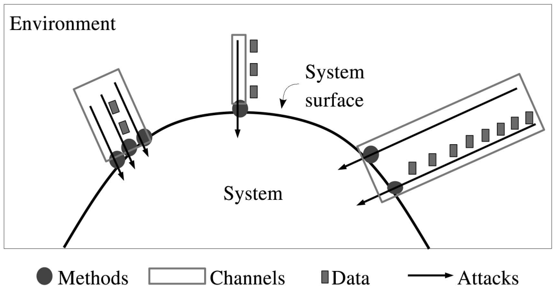

Fig. 2. A system's attack surface is the subset of the system's resources (methods, channels, and data) potentially used in attacks on the system.

sockets), invokes the system's *methods* (e.g., API), and sends (receives) *data items* (e.g., input strings) into (from) the system. An attacker can also send (receive) data indirectly into (from) a system by using *persistent* data items (e.g., files); an attacker can send data into a system by writing to a file that the system later reads. Hence, an attacker uses a system's methods, channels, and data items present in the environment to attack the system. We collectively refer to the methods, channels, and data items as the *resources* and thus define a system's attack surface in terms of the system's resources (Fig. 2).

Not all resources, however, are part of the attack surface and not all resources contribute equally to the attack surface. A resource is part of the attack surface if an attacker can use the resource to attack the system; we introduce an *entry point and exit point framework* to identify these relevant resources. A resource's contribution to the attack surface reflects the resource's likelihood of being used in attacks. For example, a method running with root privilege is more likely to be used in attacks than a method running with nonroot privilege. We introduce the notion of a *damage potential-effort ratio* to estimate a resource's contribution.

A system's attack surface measurement is the total contribution of the system's resources along the methods, channels, and data dimensions. We estimate the methods' contribution by combining the contributions of the methods that are part of the system's attack surface; we similarly estimate the contributions of the channels and the data items. A large measurement does not imply that the system has many vulnerabilities, and having few vulnerabilities does not imply a small measurement. Instead, a larger measurement indicates that an attacker is likely to exploit the vulnerabilities present in the system with less effort and cause more damage to the system. Given two systems, we compare their attack surface measurements to indicate, along each of the three dimensions, whether one is more secure than the other with respect to the attack surface metric.

The rest of the paper is organized as follows: We briefly discuss the inspiration behind our research in Section 2. In Section 3, we use an I/O automata model of a system and its environment to formalize the notion of the attack surface. In Section 4, we introduce a method for measuring the attack surfaces of systems implemented in C and apply our method to two popular open source Internet Message Access Protocol (IMAP) servers. We discuss three exploratory empirical studies for validating the attack surface metric in Section 5. In Section 6, we demonstrate that our method scales to enterprise-scale software by measuring the attack surfaces of SAP systems implemented in Java [5]. We compare our work with related work in Section 7 and conclude with a discussion of future work in Section 8.

2 Background

Our research is inspired by Michael Howard's Relative Attack Surface Quotient measurements [6]. Howard introduced the informal notion of attack surface and proposed a measurement method for the Windows operating system's (OS) attack surface. The first step in his method is identifying Windows' *attack vectors*, i.e., Windows' features often used in attacks on Windows. Examples of such features are services running on Windows, open sockets, and dynamic web pages. Not all features, however, are equally likely to be used in attacks. For example, a service running as SYSTEM is more likely to be attacked than a service running as an ordinary user. Hence, the second step in Howard's method is assigning weights to the attack vectors to reflect their *attackability*, i.e., the likelihood of a feature being used in attacks on Windows; the weights are the attack vectors' contributions to the attack surface. The final step in Howard's method is estimating the attack surface by adding the weighted counts of the attack vectors; for each instance of an attack vector, the attack vector's weight is added to the attack surface.

Howard et al. applied Howard's measurement method to seven versions of the Windows OS [7]. They identified 20 attack vectors for Windows based on the history of attacks on Windows and then assigned weights to the attack vectors based on their expert knowledge of Windows. The method was ad hoc in nature and was based on intuition; the results, however, reflected the general perception of Windows security. For example, Windows Server 2003 was perceived to have improved security compared to Windows 2000. The measurement results showed that Windows Server 2003 has a smaller attack surface than Windows 2000.

We applied Howard's measurement method to Linux to understand the challenges in applying the method [8]. We used the history of attacks on Linux to identify 14 attack vectors. Howard's method did not include any suggestion on assigning weights to attack vectors; hence, we did not assign any explicit weights. Instead, we counted the number of instances of each attack vector for four versions of Linux (three Red Hat and one Debian) and compared the numbers to get the four versions' relative attack surface measurements.

Our measurements showed that the attack surface notion held promise, e.g., Debian was perceived to be a more secure OS and that perception was reflected in our measurement. We, however, identified two shortcomings in the method. First, Howard's method is based on informal notions of a system's attack surface and attack vectors. Second, the method requires a security expert (e.g., Howard for Windows), minimally to enumerate attack vectors and assign weights to them. Thus, taken together, nonexperts cannot systematically apply his method easily.

Our research on defining a *systematic* attack surface measurement method is motivated by our above findings. We use the entry point and exit point framework to identify the relevant resources that contribute to a system's attack surface and we use the notion of the damage potential-effort ratio to estimate the weights of each such resource. Our attack surface measurement method entirely avoids the need to identify the attack vectors. Our method does not require a security expert; hence software developers with little security expertise can use the method. Furthermore, our method is applicable, not just to operating systems, but also to a wide variety of software such as web servers, IMAP servers, and application software.

3 Formal Model for a System's Attack Surface

In this section, we formalize the notion of a system's attack surface using an I/O automata model of the system and its environment [9]. We define a *qualitative* measure and a *quantitative* measure of the attack surface and introduce an *abstract method* to quantify attack surfaces.

3.1 I/O Automata Model

Informally, a system's *entry points* are the ways through which data "enter" into the system from its environment and *exit points* are the ways through which data "exit" from the system to its environment. Many attacks on software systems require an attacker either to send data into a system or to receive data from a system; hence the entry points and the exit points act as the basis for attacks on the system. We chose I/O automata as our model because our notion of entry points and exit points maps naturally to the *input actions* and *output actions* of an I/O automaton. Also, the *composition* property of I/O automata allows us to reason easily about a system's attack surface in a given environment.

An I/O automaton, $A = \langle sig(A),\ states(A), start(A),\ steps(A)\rangle$, is a four-tuple, consisting of an *action signature*, $sig(A)$, that partitions a set, $acts(A)$, of *actions* into three disjoint sets, $in(A)$, $out(A)$, and $int(A)$, of *input, output,* and *internal* actions, respectively; a set, $states(A)$, of *states*; a nonempty set, $start(A) \subseteq states(A)$, of *start states*; and a *transition relation*, $steps(A) \subseteq states(A) \times acts(A) \times states(A)$. An I/O automaton's environment generates input and transmits the input to the automaton using input actions. Conversely, the automaton generates output actions and internal actions autonomously and transmits output to its environment. Our model does not require an I/O automaton to be *input-enabled*, i.e., unlike a standard I/O automaton, input actions are not always enabled in our model. Instead, we assume that every action of an automaton is enabled in at least one reachable state of the automaton. We construct an I/O automaton modeling a complex system by *composing* the I/O automata modeling the system's simpler components. The composition of a set of I/O automata results in an I/O automaton.

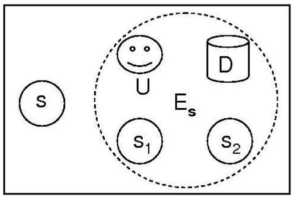

Fig. 3. A system, s, and its environment, E_s.

3.1.1 Model

Consider a set, S, of systems, a user, U, and a data store, D. For a given system, $s \in S$, we define its environment, $E_s = \langle U,\ D,\ T\rangle$, to be a three-tuple where $T = S \setminus \{s\}$ is the set of systems excluding s. s interacts with its environment E_s; hence, we define s's entry and exit points with respect to E_s. Fig. 3 shows a system, s, and its environment $E_s = \langle U,\ D,\ \{s_1,\ s_2,\}\rangle$. For example, s could be a web server and s_1 and s_2 could be an application server and a directory server, respectively.

We model every system $s \in S$ as an I/O automaton, $\langle sig(s), states(s), start(s), steps(s)\rangle$. We model the methods in s's codebase as actions of the I/O automaton. We specify the actions, using pre and postconditions: For an action, m, $m.pre$ and $m.post$ are the pre and postconditions of m, respectively. A state, $st \in states(s)$, of s is a mapping of the state *variables* to their *values*: $st : Var \rightarrow Val$. An action's pre- and postconditions are first order predicates on the state variables. A state transition, $\langle st, m, st'\rangle \in steps(s)$, is the invocation of an action m in state st, resulting in state st'. An *execution* of s is an alternating sequence of actions and states beginning with a start state and a *schedule* of an execution is a subsequence of the execution consisting only of the actions appearing in the execution.

Every system, s, has a set of *communication channels*. s's channels are the means by which the user U or any system, $s_1 \in T$, communicates with s. Specific examples of channels are sockets and named pipes. We model a system's channels as the system's special state variables.

We also model the user U and the data store D as I/O automata. U and D are global with respect to the systems in S. For simplicity, we assume only one user U present in the environment. U represents the adversary who attacks the systems in S.

We model D as a separate entity to allow sharing of data among the systems in S. D is a set of typed *data items*. Specific examples of data items are strings, URLs, files, and cookies.

3.1.2 Entry Points

A system's entry points are the methods in its codebase that receive data from the environment. The methods can receive data directly or indirectly from the environment. A method, m, of a system, s, receives data items *directly* if either 1) U (Fig. 4a) or a system, s' (Fig. 4b), in the environment invokes m and passes data items as input to m, or 2) m reads data items from D (Fig. 4c), or 3), m

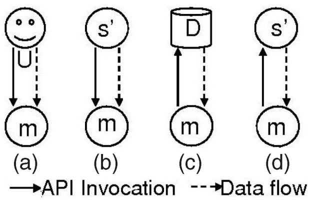

Fig. 4. Direct entry point.

invokes a method of a system s' in the environment and receives data items as results returned (Fig. 4d). A method is a *direct entry point* if it receives data items directly from the environment. Examples of a web server's direct entry points are the methods in the web server's API and the web server's methods that read configuration files.

In the I/O automata model, a system, s, can receive data from its environment if s has an input action, m, and an entity, s', in the environment has a same-named output action m. When s' performs its output action m and s performs its input action m, data are transmitted from s' to s. We formalize the scenarios when s' invokes m (Fig. 4b) or when m invokes s''s method (Fig. 4d) the same way, i.e., s has an input action, m, and s' has an output action m.

Definition 1. *A* direct entry point *of a system s is an input action, m, of s, such that either 1) U has the output action m (Fig. 4a), or 2) a system $s' \in T$ has the output action m (Figs. 4b and 4d), or 3) D has the output action m (Fig. 4c).*

A method, m, of s receives data items *indirectly* if either 1) a method, m_1, of s receives a data item, d, directly and either m_1 passes d as input to m (Fig. 5a) or m receives d as result returned from m_1 (Fig. 5b), or 2) a method, m_2, of s receives a data item, d, indirectly, and either m_2 passes d as input to m (Fig. 5c) or m receives d as result returned from m_2 (Fig. 5d). A method is an *indirect entry point* if it receives data items indirectly from the environment. For example, if a web server's API method, m, receives login information from users and passes the information to an authentication module method, m_1, then m is a direct entry point and m_1 is an indirect entry point.

In the I/O automata model, a system's internal actions are not visible to other systems in the environment. Hence we use internal actions to formalize indirect entry points. We formalize data transmission using actions' pre and postconditions. If an input action, m, of a system, s, receives a data item, d, directly from the environment, then s's subsequent behavior depends on d; hence d appears in the postcondition of m and we write $d \in Res(m.post)$ where

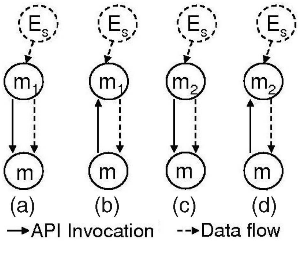

Fig. 5. Indirect entry point.

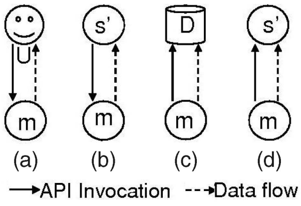

Fig. 6. Direct exit point.

$Res : predicate \rightarrow 2^{Var}$ is a function such that for each postcondition (or precondition), p, $Res(p)$ is the set of resources appearing in p. Similarly, if an action, m, receives a data item, d, from another action, m_1, then d appears in m_1's postcondition and in m's precondition. Similar to the direct entry points, we formalize the scenarios Figs. 5a and 5b the same way and the scenarios Figs. 5c and 5d the same way. We define indirect entry points recursively.

Definition 2. *An* indirect entry point *of a system, s, is an internal action, m, of s such that either 1) $\exists$ direct entry point, m_1, of s such that $m_1.post \Rightarrow m.pre$ and $\exists$ a data item, d, such that $d \in Res(m_1.post) \wedge d \in Res(m.pre)$ (Figs. 5a and 5b), or 2) $\exists$ indirect entry point, m_2, of s such that $m_2.post \Rightarrow m.pre$ and $\exists$ data item, d, such that $d \in Res(m_2.post) \wedge d \in Res(m.pre)$ (Figs. 5c and 5d).*

3.1.3 Exit Points

A system's methods that send data to its environment are the system's exit points. For example, a method that writes to a log file is an exit point. The methods can send data directly or indirectly to the environment. A method, m, of a system, s, sends data *directly* if either 1) U (Fig. 6a) or a system, s' (Fig. 6b), in the environment invokes m and receives data items as results returned from m, or 2) m writes data items to D (Fig. 6c), or 3) m invokes a method of a system, s', in the environment and passes data items as input (Fig. 6d).

In the I/O automata model, a system, s, can send data to the environment if s has an output action, m and an entity s' in the environment has a same-named input action, m. When s performs its output action m, s' performs its input action m, and data are transmitted from s to s'.

Definition 3. *A* direct exit point *of a system, s, is an output action, m, of s such that either 1) U has the input action m (Fig. 6a), or 2) a system, $s' \in T$, has the input action m (Figs. 6b and 6d), or 3) D has the input action m (Fig. 6c).*

A method, m, of s sends data items *indirectly* to the environment if either 1) m passes a data item, d, as input to a direct exit point, m_1 (Fig. 7a), or m_1 receives a data item, d, as result returned from m (Fig. 7b), and m_1 sends d directly to the environment, or 2) m passes a data item, d, as input to an indirect exit point m_2 (Fig. 7c), or m_2 receives a data item, d, as result returned from m (Fig. 7d), and m_2 sends d indirectly to the environment. A method, m, of s is an *indirect exit point* if m sends data items indirectly to the environment.

Similar to indirect entry points, we formalize indirect exit points using an I/O automaton's internal actions. Again, we define indirect exit points recursively.

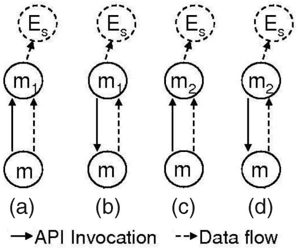

Fig. 7. Indirect exit point.

Definition 4. *An* indirect exit point *of a system, s, is an internal action, m, of s such that either 1) $\exists$ a direct exit point, m_1, of s such that $m.post \Rightarrow m_1.pre$ and $\exists$ a data item, d, such that $d \in Res(m.post) \wedge d \in Res(m_1.pre)$ (Figs. 7a and 7b), or 2) $\exists$ an indirect exit point, m_2, of s such that $m.post \Rightarrow m_2.pre$ and $\exists$ a data item, d, such that $d \in Res(m.post) \wedge d \in Res(m_2.pre)$ (Figs. 7c and 7d).*

3.1.4 Channels

An attacker uses a system's channels to connect to the system and invoke the system's methods. Hence, the channels act as another basis for attacks on the system. An entity in the environment can invoke a method, m, of a system, s, by using s's channel, c; hence, in our I/O automata model, c appears in a direct entry point (or exit point), m's, precondition, i.e., $c \in Res(m.pre)$. In our model, every channel must appear in at least one direct entry point's (or direct exit point's) precondition. Similarly, at least one channel must appear in every direct entry point's (or direct exit point's) precondition.

3.1.5 Untrusted Data Items

An attacker can use persistent data items to send (receive) data indirectly into (from) a system. Hence, the persistent data items act as another basis of attacks on a system. The data items which are visible to both a system, s, and U across s's different executions are s's persistent data items. Specific examples of persistent data items are files, cookies, database records, and registry entries. The persistent data items are shared between s and U. For example, s might read a file after U writes to the file. An *untrusted data item* of a system, s, is a persistent data item, d, such that a direct entry point of s reads d from the data store or a direct exit point of s writes d to the data store.

Definition 5. *An* untrusted data item *of a system, s, is a persistent data item, d, such that either 1) $\exists$ a direct entry point, m, of s such that $d \in Res(m.post)$, or 2) $\exists$ a direct exit point, m, of s such that $d \in Res(m.pre)$.*

3.1.6 Attack Surface Definition

A system's attack surface is the subset of its resources that an attacker can use to attack the system. An attacker can use a system's entry points and exit points, channels, and untrusted data items to send (receive) data into (from) the system to attack the system. Hence the set of entry points and exit points, the set of channels, and the set of untrusted data items are the relevant subset of resources that are part of the attack surface.

Definition 6. *Given a system, s, and its environment, E_s, s's attack surface is the triple, $\langle M^{E_s}, C^{E_s}, I^{E_s} \rangle$, where M^{E_s} is s's set of entry points and exit points, C^{E_s} is s's set of channels, and I^{E_s} is s's set of untrusted data items.*

Notice that we define s's entry points and exit points, channels, and data items with respect to the given environment E_s. Hence s's attack surface, $\langle M^{E_s}, C^{E_s}, I^{E_s} \rangle$, is with respect to the environment E_s. We compare the attack surfaces of two similar systems (i.e., different versions of the same software or different software that provide similar functionality) along the methods, channels, and data dimensions with respect to the same environment to determine if one has a larger attack surface than another.

Definition 7. *Given an environment, $E = \langle U, D, T \rangle$, and systems, A and B, A's attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, is larger than B's attack surface, $\langle M_B^E, C_B^E, I_B^E \rangle$, iff either 1) $M_A^E \supset M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supseteq I_B^E$, or 2) $M_A^E \supseteq M_B^E \wedge C_A^E \supset C_B^E \wedge I_A^E \supseteq I_B^E$, or 3) $M_A^E \supseteq M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supset I_B^E$.*

We model s's interactions with the entities present in its environment as parallel composition, $s \| E_s$. Notice that an attacker can send data into s by invoking s's input actions and can receive data from s when s executes its output actions. Since an attacker attacks a system by sending (receiving) data into (from) the system, any schedule of $(s \| E_s)$ that contains s's input actions or output actions is a potential attack on s. We denote the set of potential attacks on s as $attacks(s)$.

Definition 8. *Given a system, s, and its environment, $E_s = \langle U, D, T \rangle$, a* potential attack *on s is a schedule, β, of the composition, $P = s \parallel U \parallel D \parallel (\|_{t \in T}\, t)$, such that an input action (or output action), m, of s appears in β.*

We show that with respect to the same attacker and operating environment, if a system, A, has a larger attack surface than a similar system, B, then the number of potential attacks on A is larger than B (we omit the proof due to space limitations [10]). Since A and B are similar systems, i.e., different versions of the same system (e.g., different versions of the Windows OS) or different systems with similar functionality (e.g., different File Transfer Protocol (FTP) servers), we assume that both A and B have the same set of state variables and the same set of resources except the ones appearing in the attack surfaces.

Theorem 1. *Given an environment, $E = \langle U, D, T \rangle$, and systems, A and B, if A's attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, is larger than B's attack surface, $\langle M_B^E, C_B^E, I_B^E \rangle$, and the rest of the resources of A and B are equal, then $attacks(A) \supset attacks(B)$.*

Theorem 1 has practical significance in the software development process. The theorem shows that if we create a newer version of a software system by only adding more resources to an older version, then assuming all resources are counted equally (see Section 3.2), the newer version has a larger attack surface, and hence, a larger number of potential attacks. Software developers should ideally strive toward reducing the attack surface of their software from one version to another, or if adding resources to the software (e.g., adding methods to an API), then do so knowing that they are increasing the attack surface.

3.2 Damage Potential and Effort

Not all resources contribute equally to a system's attack surface measurement, because not all resources are equally likely to be used by an attacker. A resource's contribution to a system's attack surface depends on the resource's *damage potential*, i.e., the level of harm the attacker can cause to the system in using the resource in an attack and the *effort* the attacker spends to acquire the necessary access rights in order to be able to use the resource in an attack. The higher the damage potential or the lower the effort, the higher the resource's contribution to the attack surface. In this section, we use our I/O automata model to formalize the notions of damage potential and effort. We model a resource, r's, damage potential and effort as the state variables, $r.dp$ and $r.ef$, respectively.

In practice, we estimate a resource's damage potential and effort in terms of the resource's attributes, e.g., method privilege, access rights, channel protocol, and data item type. Our estimation method is a specific instantiation of our general measurement framework and includes only technical impact (e.g., privilege elevation) and not business impact (e.g., monetary loss) though our framework does not preclude this generality. We do not make any assumptions about the attacker's capabilities or resources in estimating damage potential or effort.

We estimate a method's damage potential in terms of the method's *privilege*. An attacker gains a method's privilege by using the method in an attack. For example, the attacker gains `root` privilege by exploiting a buffer overflow in a method running as `root`. The attacker can cause damage to the system after gaining `root` privilege. The attacker uses channels to connect a system and send (receive) data to (from) the system. A channel's *protocol* imposes restrictions on the data exchange allowed using the channel, e.g., a `TCP socket` allows raw bytes to be exchanged whereas an `RPC endpoint` does not. Hence we estimate a channel's damage potential in terms of its protocol. The attacker uses persistent data items to send (receive) data indirectly into (from) a system. A persistent data item's *type* imposes restrictions on the data exchange, e.g., a `file` can contain executable code whereas a `registry entry` cannot. The attacker can send executable code into the system by using a `file`, but the attacker cannot do the same using a `registry entry`. Hence we estimate a data item's damage potential in terms of the its type. The attacker can use a resource in an attack if the attacker has the required *access rights*. The attacker spends effort to acquire these access rights. Hence for the three kinds of resources, i.e., method, channel, and data, we estimate attacker effort in terms of the resource's access rights.

We assume that we have a total ordering, $\succ$, among the six attributes' values, i.e., method privilege and access rights, channel protocol and access rights, and data item type and access rights. In practice, we impose these total orderings using our knowledge of a system and its environment. For example, an attacker can cause more damage to a system by using a method running with `root` privilege than a method running with `nonroot` privilege; hence `root` $\succ$ `nonroot`. We use these total orderings to compare the contributions of resources to the attack surface. Abusing notation, we write $r_1 \succ r_2$ to express that a resource, r_1, makes a larger contribution than a resource, r_2.

Definition 9. *Given two resources, r_1 and r_2, $r_1 \succ r_2$ iff either 1) $r_1.dp \succ r_2.dp \wedge r_2.ef \succ r_1.ef$, or 2) $r_1.dp = r_2.dp \wedge r_2.ef \succ r_1.ef$, or 3) $r_1.dp \succ r_2.dp \wedge r_2.ef = r_1.ef$.*

Definition 10. *Given two resources, r_1 and r_2, $r_1 \succeq r_2$ iff either 1) $r_1 \succ r_2$ or 2) $r_1.dp = r_2.dp \wedge r_2.ef = r_1.ef$.*

3.2.1 Modeling Damage Potential and Effort

In our I/O automata model, we use an action's pre and postconditions to formalize effort and damage potential, respectively. Intuitively, the effort corresponds to the preconditions the attacker needs to satisfy to invoke an action and the damage potential corresponds to the action invocation's damaging effect stated in the action's postcondition. We present a parametric definition of an action, m, of a system, s, below. For readability, we show this definition for when the entities in the environment connect to s using only one channel, c, to invoke m and m either reads or writes only one data item, d. The generalization to a vector of channels and a vector of data items is straightforward.

$$\begin{aligned}
&m(MA, CA, DA, MB, CB, DB)\\
&pre: P_{pre} \wedge MA \succeq m.ef \wedge CA \succeq c.ef \wedge DA \succeq d.ef\\
&post: P_{post} \wedge MB \succeq m.dp \wedge CB \succeq c.dp \wedge DB \succeq d.dp.
\end{aligned}$$

The parameters MA, CA, and DA represent the highest method access rights, channel access rights, and data access rights acquired by an attacker so far, respectively. Similarly, MB, CB, and DB represent the benefit to the attacker in using m, c, and d in an attack, respectively. P_{pre} is the part of m's precondition that does not involve access rights. The clause, $MA \succeq m.ef$, captures the condition that the attacker has the required access rights to invoke m; the other two clauses in the precondition are analogous. Similarly, P_{post} is the part of m's postcondition that does not involve benefit. The clause, $MB \succeq m.dp$, captures the condition that the attacker gets the expected benefit after the execution of m; the rest of the clauses are analogous.

3.2.2 Attack Surface Measurement

Given two systems, A and B, if A has a larger attack surface than B (Definition 7), then everything else being equal, it is easy to see that A has a larger attack surface measurement than B. It is also possible that even though A and B both have the same attack surface, if a resource, $A.r$, belonging to A's attack surface makes a larger contribution than the same-named resource, $B.r$, belonging to B's attack surface, then everything else being equal A has a larger attack surface measurement than B.

Given a system, A, and its attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, we denote the set of resources belonging to A's attack surface as $R_A = M_A^E \cup C_A^E \cup I_A^E$. Note that from Definition 7, if A has a larger attack surface than B, then $R_A \supset R_B$.

Definition 11. *Given an environment, $E = \langle U, D, T \rangle$, systems, A and B, A's attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, and*

B's attack surface, $\langle M_B^E, C_B^E, I_B^E \rangle$, A has a larger attack surface measurement than B iff either

1. *A has a larger attack surface than B (i.e., $R_A \supset R_B$) and $\forall r \in R_B.\ A.r \succeq B.r$, or*
2. *$M_A^E = M_B^E \ \wedge\ C_A^E = C_B^E \ \wedge\ I_A^E = I_B^E$ (i.e., $R_A = R_B$) and there is a nonempty set, $\mathcal{R}_{\mathcal{AB}} \subseteq R_B$, of resources such that $\forall r \in \mathcal{R}_{\mathcal{AB}}.\ A.r \succ B.r$ and $\forall r \in (R_B \setminus \mathcal{R}_{\mathcal{AB}}).\ A.r = B.r$.*

We show that with respect to the same attacker and operating environment, if a system, A, has a larger attack surface measurement than a system, B, then the number of potential attacks on A is larger than B (we omit the proof due to space limitations [10]).

Theorem 2. *Given an environment, $E = \langle U, D, T \rangle$, systems A and B, A's attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, and B's attack surface, $\langle M_B^E, C_B^E, I_B^E \rangle$, if A has a larger attack surface measurement than B, then $attacks(A) \supseteq attacks(B)$.*

Theorem 2 also has practical significance in the software development process. The theorem shows that if software developers increase a resource's damage potential and/or decrease the resource's effort in their software's newer version, then all else being equal, the newer version's attack surface measurement becomes larger and the number of potential attacks on the software increases.

3.3 A Quantitative Metric

The qualitative attack surface measurement introduced in Definition 11 is useful to determine if one system has a larger attack surface measurement than another. We, however, need a quantitative measure to determine the difference in the measurements. In this section, we introduce a quantitative attack surface measurement in terms of the resources' *damage potential-effort ratios*.

3.3.1 Damage Potential-Effort Ratio

We considered damage potential and effort in isolation while estimating a resource's contribution to the attack surface. From an attacker's point of view, however, damage potential and effort are related; if the attacker gains higher privilege by using a method in an attack, then the attacker also gains the access rights of a larger set of methods. For example, the attacker can access only the methods with `authenticated` user access rights by gaining `authenticated` privilege, whereas the attacker can access methods with `authenticated` user and `root` access rights by gaining `root` privilege. The attacker might be willing to spend more effort to gain a higher privilege level, which then enables the attacker to cause damage as well as gain more access rights. Hence we consider damage potential and effort in tandem and quantify a resource's contribution as a damage potential-effort ratio. The ratio is similar to a cost-benefit ratio; the damage potential is the benefit to the attacker in using a resource in an attack and the effort is the cost to the attacker in using the resource.

In our I/O automata model, a method, m's, damage potential determines the potential number of methods that m can call and hence the potential number of methods that can follow m in a schedule; the higher the damage potential, the larger the number of methods. Similarly, m's effort determines the potential number of methods that can call m and hence, the potential number of methods that m can follow in a schedule; the lower the effort, the larger the number of methods. Hence m's damage potential-effort ratio $der_m(m)$ determines the potential number of schedules in which m can appear. Given two methods, m_1 and m_2, if $der_m(m_1) > der_m(m_2)$, then m_1 can potentially appear in more schedules (and hence, more potential attacks) than m_2. Similarly, if a channel, c (or a data item, d), appears in a method m's precondition, then c's damage potential-effort ratio (or d's) determines the potential number of schedules in which m can appear. Hence, we estimate a resource's contribution to the attack surface as the resource's damage potential-effort ratio.

We assume a function, $der_m : \text{method} \rightarrow \mathbb{Q}$, that maps each method to its damage potential-effort ratio belonging to the set $\mathbb{Q}$ of rational numbers. Similarly, we assume a function, $der_c : \text{channel} \rightarrow \mathbb{Q}$ for the channels and a function, $der_d : \text{data item} \rightarrow \mathbb{Q}$ for the data items. In practice, however, we compute a resource's damage potential-effort ratio by assigning numeric values to the resource's attributes. We discuss a specific numeric value assignment method in Section 4.2.

3.3.2 Quantitative Attack Surface Measurement Method

We quantify a system's attack surface measurement along three dimensions: methods, channels, and data.

Definition 12. *Given a system, s's, attack surface, $\langle M^{E_s}, C^{E_s}, I^{E_s} \rangle$, s's attack surface measurement is the triple $\langle \sum_{m \in M^{E_s}} der_m(m),\ \sum_{c \in C^{E_s}} der_c(c),\ \sum_{d \in I^{E_s}} der_d(d) \rangle$.*

Our attack surface measurement method is analogous to the risk estimation method used in risk modeling [11]. s's attack surface measurement is an indication of s's risk from attacks on s. In risk modeling, the risk associated with a set E of events is $\sum_{e \in E} p(e)C(e)$, where an event, e's, occurrence probability is $p(e)$ and consequence is $C(e)$. The events in risk modeling are analogous to the resources in our measurement method. An event's occurrence probability is analogous to the probability of a successful attack using a resource. For example, a buffer overrun attack using a method, m, will be successful only if m has an exploitable buffer overrun vulnerability. Hence, the probability, $p(m)$, associated with m is the probability that m has an exploitable vulnerability. Similarly, the probability, $p(c)$, associated with a channel, c, is the probability that the method that receives (or sends) data from (to) c has an exploitable vulnerability and the probability, $p(d)$, associated with a data item, d, is the probability that the method that reads or writes d has an exploitable vulnerability. An event's consequence is analogous to a resource's damage potential-effort ratio. The pay-off to the attacker in using a resource in an attack is proportional to the resource's damage potential-effort ratio; hence the damage potential-effort ratio is the consequence of a resource being used in an attack. The risk along s's three dimensions is the triple, $\langle \sum_{m \in M^{E_s}} p(m) der_m(m), \sum_{c \in C^{E_s}} p(c) der_c(c), \sum_{d \,\in\, I^{E_s}} p(d) der_d(d) \rangle$, which is also s's attack surface measurement.

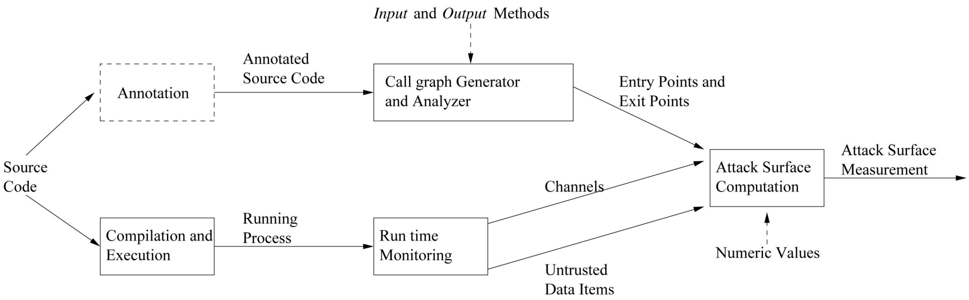
Input and *Output* Methods
Annotation
Annotated Source Code
Call graph Generator and Analyzer
Entry Points and Exit Points
Source Code
Attack Surface Computation
Attack Surface Measurement
Channels
Compilation and Execution
Running Process
Run time Monitoring
Numeric Values
Untrusted Data Items

Fig. 8. Steps of our attack surface measurement method for C.

In practice, however, predicting defects in software [12] and estimating the likelihood of vulnerabilities in software are difficult tasks [13]. Hence we take a conservative approach in our measurement method and assume that $p(m) = 1$ for all methods, i.e., every method has an exploitable vulnerability; even if a method does not have a known vulnerability now, it might have a future vulnerability not discovered so far. We similarly assume that $p(c) = 1$ for all channels and $p(d) = 1$ for all data items. With our conservative approach, s's attack surface measurement is $\langle \sum_{m \in M^{E_s}} der_m(m), \sum_{c \in C^{E_s}} der_c(c), \sum_{d \in I^{E_s}} der_d(d) \rangle$.

4 Empirical Attack Surface Measurements

In this section, we instantiate the previous section's abstract measurement method for systems implemented in the C programming language; Fig. 8 shows the steps in the instantiated method. The dotted box shows the step done manually and the solid boxes show the steps done programmatically. The two dotted arrows represent manual inputs required for measuring the attack surface. We automated as many steps as possible in our measurement method and minimized the number of manual inputs required by the method.

The empirical application of our method focuses only on direct entry and exit points since, unlike for indirect ones, we can automatically identify direct ones using source code analysis tools. Developing automated tools to identify indirect entry and exit points whose use would then afford a more complete attack surface measurement is left for future work.

We illustrate the method by describing the measurement process of two popular open source IMAP servers: Courier-IMAP 4.0.1 and Cyrus 2.2.10. We also measured the attack surfaces of two FTP daemons: ProFTPD 1.2.10 and Wu-FTPD 2.6.2; we omit the results due to space limitations [10]. We chose the IMAP servers due to their popularity and measured the attack surfaces of the IMAP daemons in both codebases to obtain a fair comparison.

4.1 Identification of Entry Points and Exit Points, Channels, and Untrusted Data Items

A system's direct entry point (exit point) is a method that receives (sends) data from (to) the environment. As proposed by DaCosta et al., we assume that a method can receive (send) data from (to) the environment by invoking specific C library methods, e.g., the `read` method defined in `unistd.h` [14]; we identify a set *Input* (*Output*) of relevant C library methods. A method is a direct entry point (exit point) if it contains a call to a method in *Input* (*Output*). Hence we automatically construct a call graph from a system's source code and identify all methods that contain calls to methods in *Input* (*Output*) as direct entry points (exit points).

On a UNIX system, a process changes its privilege through a set of *uid-setting* system calls such as `setuid` [15]. Hence we determine entry (exit) points' privileges by locating uid-setting system calls in the code base. For example, if a process starts with `root` privilege and then drops privilege by calling `setuid`, then we assume all methods invoked before `setuid` to have `root` privilege and all methods invoked after `setuid` to have `nonroot` privilege. Similarly, we identify entry points' access rights from the code location where authentication is performed. We assume that any method that is invoked before user authentication takes place has unauthenticated access rights and any method that is invoked after successful authentication has authenticated access rights.

We generated call graphs for both daemons using `cflow` [16]. If a method ran with multiple privileges or was accessible with multiple access rights levels during different executions, then we counted the method multiple times. We show the direct entry points (DEP) and direct exit points (DExP) in Table 1; please note that all methods in the Cyrus codebase run with a special UNIX user, `cyrus`, and privilege.

We observed the run time behavior of both daemons' default installations to identify the daemons' channels, untrusted data items, and their relevant attributes. We show the daemons' channels in Table 2 and untrusted data items in Table 3.

4.2 Estimation of Damage Potential-Effort Ratio

We assign numeric values to the six attributes introduced in Section 3.2 to estimate numeric damage potential-effort

TABLE 1
IMAP Daemons' Entry Points and Exit Points

Privilege	Access Rights	DEP	DExP
	Courier		
`root`	`unauthenticated`	28	17
`root`	`authenticated`	21	10
`authenticated`	`authenticated`	113	28
	Cyrus		
`cyrus`	`unauthenticated`	16	17
`cyrus`	`authenticated`	12	21
`cyrus`	`admin`	13	22
`cyrus`	`anonymous`	12	21

TABLE 2
IMAP Daemons' Channels

Type	Access Rights	Count
Courier		
TCP	remote unauthenticated	1
SSL	remote unauthenticated	1
UNIX socket	local authenticated	1
Cyrus		
TCP	remote unauthenticated	2
SSL	remote unauthenticated	1
UNIX socket	local authenticated	1

ratios. We impose a total order among the values of the attributes and assign numeric values according to the total order. For example, we assume a method running as `root` has a higher damage potential than a method running as `authenticated` user; hence, `root > authenticated` user in the total ordering and we assign a higher number to `root` than `authenticated` user. The exact choice of the numeric values is subjective and depends on a system and its environment. Hence we cannot automate the process of numeric value assignment. We, however, provide guidelines to our users for numeric value assignment using parameter sensitivity analysis.

In our parameter sensitivity analysis, we studied the effects of changing the difference in the numeric values assigned to the attributes on our measurements. Numeric values should be chosen such that both the privilege values and the access rights values affect the attack surface measurements comparison's outcome. Our analysis shows that if both systems have comparable numbers of entry points and exit points, then the access rights values do not affect the measurements if the privilege difference is low or high. Similarly, if one system has a significantly larger number of entry points and exit points than the other, then no choice of the privilege difference or the access rights difference affects the measurement. Please see Section 4.5 of [10] for further details on the parameter sensitivity analysis.

In the case of the IMAP daemons, we assigned numeric values based on our knowledge of the IMAP servers and UNIX security (Table 4). Please note a few special orderings among the attributes: A method running with `cyrus` privilege in the Cyrus daemon has access to every user's email files; hence, we assumed a method running as `cyrus` has higher damage potential than a method running as `authenticated` user. Similarly, `admin` and `cyrus` are special users in Cyrus; hence, we assumed the attacker spends greater effort to acquire `admin` and `cyrus` access rights compared to `authenticated` access rights. We also assumed that each channel has the same damage potential.

TABLE 3
IMAP Daemons' Untrusted Data Items

Type	Access Rights	Count
Courier		
file	root	74
file	authenticated	13
file	world	53
Cyrus		
file	root	50
file	cyrus	26
file	world	50

TABLE 4
Numeric Values Assigned to the Attributes

Method Privilege	Value	Access Rights	Value
root	5	admin	4
cyrus	4	authenticated	3
authenticated	3	anonymous	1
		unauthenticated	1
Channel Type	Value	Access Rights	Value
TCP	1	local auth	4
SSL	1	remote unauth	1
UNIX socket	1		
Data Item Type	Value	Access Rights	Value
file	1	root	5
		cyrus	4
		authenticated	3
		world	1

4.3 Attack Surface Measurements and Their Usage

We estimated the methods' total contribution, the channels' total contribution, and the data items' total contribution to both IMAP daemon's attack surfaces. For example, from Table 1 and Table 4, Courier's methods contribute $(45 \times (\frac{5}{1}) + 31 \times (\frac{5}{3}) + 141 \times (\frac{3}{3})) = 417.67$. From similar computations, Courier's attack surface measurement is $\langle 417.67, 2.25, 72.13 \rangle$ and Cyrus's attack surface measurement is $\langle 343.00, 3.25, 66.50 \rangle$.

The attack surface metric tells us that the Cyrus daemon presents less security risk along the methods and data dimensions, whereas the Courier daemon presents less security risk along the channels dimension. Keeping the measurement separated along three different dimensions offers a design choice to our users, e.g., system administrators can choose a dimension appropriate for their need. In order to choose one daemon over another, we first determine the dimension that presents more risk using our knowledge of the daemons and the operating environment; we then make a choice using the measurements along that dimension. For example, if we were concerned about privilege elevation on the host running the IMAP daemon, then the methods dimension presents more risk and the attack surface metric suggests that we would choose the Cyrus daemon over the Courier daemon. Similarly, if we were concerned about the number of open channels on the host running the IMAP daemon, then the channels dimension presents more risk and we would choose the Courier daemon. If we were concerned about the safety of email files, then the data dimension presents more risk and we would choose the Cyrus daemon.

5 Empirical Studies for Validation

A key challenge in security metrics research is the validation of a metric. Validating a software attribute's measure is hard [17], [18], [19]; security is an attribute that is hard to measure and hence even harder to validate. To validate our metric, we conducted three exploratory empirical studies inspired by the research community's software metrics validation approaches [20], [21].

In practice, validation approaches distinguish *measures* from *prediction systems*; measures numerically characterize software attributes whereas prediction systems predict software attributes' values. For example, lines of code (LOC) is a measure of software "length"; LOC becomes a prediction system if we use LOC to predict software "complexity." We validate a measure by showing its correctness in numerically characterizing an attribute and a prediction system by showing its accuracy. Our attack surface metric plays a dual role: It measures a software attribute, i.e., the attack surface, and is also a prediction system to indicate software's security risk. Hence we took a two-fold validation approach. First, we validated the measure by validating our measurement method using two empirical studies: a statistical analysis of data collected from Microsoft Security Bulletins (Section 5.1) and an expert user survey for Linux (Section 5.2). Our approach is motivated by the notion of *convergent evidence* in Psychology [22]; since each study has its own strengths and weaknesses, the convergence in the studies' findings enhances our belief that the findings are valid and not methodological artifacts.

Second, we validated our metric's prediction system by validating attack surface measurements. In Section 3, we formally showed that a larger attack surface leads to a larger number of potential attacks on software. We established a relationship between attack surface measurements and security risk by analyzing vulnerability patches in open source software (Section 5.3). We also gathered anecdotal evidence from software industry to show that attack surface reduction mitigates security risk (Section 5.4).

Liu and Traore introduce a theoretical validation framework, based on established security design principles, to validate security metrics [23]. They demonstrate that our metric is valid in their framework; their theoretical validation complements our empirical studies.

5.1 Statistical Analysis of Microsoft Security Bulletins (MSB)

Our measurement method is based on the following three key hypotheses; hence we validated the hypotheses to validate the method.

1. Methods, channels, and data are the attack surface's dimensions.
2. The six resource attributes (method privilege and access rights, channel protocol and access rights, and data item type and access rights) are indicators of damage potential and effort.
3. A resource's damage potential-effort ratio is an indicator of the resource's likelihood of being used in attacks.

5.1.1 MSBs

An MSB describes an exploitable vulnerability present in Microsoft software [24]. We collected data from 110 bulletins, published over a period of two years, by manually interpreting the bulletins' descriptions; hence the process is subject to human error. We identified the resources (methods, channels, and data items) that the attacker has to use to exploit the vulnerabilities described in the bulletins, the resources' attributes that are indicators of damage potential and effort, and the bulletins' severity ratings. Many bulletins contained multiple vulnerabilities; hence the 110 bulletins resulted in 202 observations.

5.1.2 Hypothesis 1

Out of the 202 observations, 202 mention methods, 170 mention channels, and 108 mention data items as the resources used in exploiting the vulnerabilities. These findings suggest that methods, channels, and data items are used in attacks on software and hence are the attack surface's dimensions. We, however, cannot rule out other dimensions, even though we did not find any other resource types mentioned in the bulletins.

TABLE 5
Significance of Method and Channel Attributes

Attribute	Coefficient	Standard Error	p-value
Methods			
Privilege	0.948	0.236	$p < 0.001$
Access Rights	-0.584	0.110	$p < 0.001$
Channels			
SMTP	2.535	0.504	$p < 0.001$
TCP	0.957	0.466	$p = 0.040$
Pipe	0.948	0.574	$p = 0.099$
Access Rights	-0.312	0.109	$p = 0.004$

5.1.3 Hypothesis 2

A bulletin's severity rating depends on Microsoft's assessment of the impact of exploiting the vulnerability described in the bulletin and the exploitation's difficulty. The higher the impact and the lower the difficulty, the higher the rating. The exploitation's impact and difficulty are equivalent to damage potential and attacker effort in our measurement method, respectively. Hence we expect the severity rating to depend on the six attributes that are indicators of damage potential and attacker effort; we also expect the severity rating to increase with an increase in the value of an attribute that is an indicator of damage potential, e.g., method privilege, and to decrease with an increase in the value of an attribute that is an indicator of attacker effort, e.g., method access rights. In other words, we expect an indicator of damage potential (effort) to be a significant predictor of the severity rating and to be positively (negatively) correlated with the severity rating.

We used ordered logistic regression to test for the attributes' significance as logistic regression uses maximum likelihood estimates to compute the regression coefficients. A positive coefficient indicates a positive correlation between an attribute and the severity rating, and a negative coefficient indicates a negative correlation. We used z-tests to determine the coefficients' statistical significance (p-value < 0.05); our null hypothesis was that the coefficients are zero and hence the attributes are not significant predictors of the severity rating.

Following the process described in Section 4.2, we assigned numeric values to the six attributes and the severity rating to generate a data set for performing ordered logistic regression. We imposed total orderings among the method privileges, method access rights, channel access rights, data item access rights, and the severity ratings, and we assigned numeric values according to the total ordering and on an ordinal scale. We, however, could not impose total orderings among channel protocols and file formats. Hence, we assigned numeric values on a nominal scale. Since nominal values are not ordered, we could not determine a positive or negative correlation with severity rating.

Tables 5 and 6 show our results that suggest that the six attributes are indicators of damage potential and effort. A method's privilege (Table 5, row 3), a method's access rights (Table 5, row 4), a channel's access rights (Table 5, row 9),

TABLE 6
Significance of Data Item Attributes

Data Items			
Attribute	Coefficient	Standard Error	p-value
HTML	-0.651	0.263	p = 0.013
DHTML	-0.589	0.437	p = 0.177
ActiveX	1.522	0.480	p = 0.002
WMF	46.314	2.58e+09	p = 1.000
Doc	-1.123	0.462	p = 0.015
Access Rights	-0.310	0.078	p $<$ 0.001

and a data item's access rights (Table 6, row 8) are significant predictors of the severity rating and exhibit expected correlation with the severity rating. Table 5 shows that `SMTP` and `TCP` are significant and `pipe` is insignificant in explaining the severity rating. Since two of the three protocols are significant, the finding suggests that channel protocol is a significant predictor of the severity rating. Similarly, Table 6 shows that data item type is a significant predictor of the severity rating.

5.1.4 Hypothesis 3

The bulletins did not have any data relevant to a resource's likelihood of being used in attacks. Henc, we could not use the MSBs to validate Hypothesis 3. We used an expert survey described in Section 5.2 to validate Hypothesis 3.

5.2 Expert User Survey

Statistical survey is a widely used technique in social sciences [25], [26] and is also used for empirical validation in software engineering research [27], [28], [29]. We conducted an expert user survey for two reasons: First, we wanted to find out potential users' perception of our metric. Second, our MSB analysis was with respect to Windows; we conducted the survey with respect to Linux.

5.2.1 Subjects and Questionnaire

Software developers and software consumers are our metric's two potential user groups. We collaborated with software developers and got their feedback on improving the metric; we discuss the collaboration in Section 6. System administrators are examples of software consumers; hence we conducted an e-mail survey of expert Linux administrators to know their perception of the metric. We identified 20 experienced system administrators working in universities, corporations, and government agencies as our survey's subjects.

The survey questionnaire consisted of six explanatory questions designed to measure the subjects' attitude about our measurement method's steps. The first five questions asked the subjects to indicate their degree of agreement or disagreement with the steps. We used the last question to collect information about the subjects to avoid self-selection bias, i.e., the subjects incorrectly consider themselves expert system administrators without relevant experience or expertise. The subjects indicated their attitude on a five-point Likert scale: strongly agree, agree, neither agree nor disagree, disagree, and strongly disagree [30]. The Likert scale's bipolar scaling nature allows us to measure both positive and negative responses. We conducted six rounds of pretesting and post-test interviews to identify and remove leading questions, ambiguous terms, and overall confusing questions from the questionnaire.

TABLE 7
The Subjects' Perception about the Dimensions and the Damage Potential-Effort Ratio

	Agree	Disagree	Neutral	p-value
Dimensions				
Methods	95%	0%	5%	p $<$ 0.0001
Channels	95%	0%	5%	p $<$ 0.0001
Data	85%	0%	15%	p $<$ 0.0001
Damage Potential-Effort Ratio (der)				
der	70%	20%	10%	p = 0.0141

5.2.2 Results

We combined the "strongly agree" and the "agree" responses to an "agree (strongly or otherwise)" category and the "strongly disagree" and the "disagree" responses to a "disagree (strongly or otherwise)" category to avoid central tendency bias, i.e., subjects may avoid using extreme response categories such as "strongly agree." We then computed the proportion of the subjects who agree with, disagree with, and are neutral with our method's steps. We performed t-tests to determine the survey responses' statistical significance (p-value < 0.05); we used the null hypothesis that the mean of a survey question's Likert scale responses is "neutral."

A majority of the subjects agreed with our choice of the attack surface's three dimensions (Table 7) and our notion of the damage potential-effort ratio as an indicator of a resource's likelihood of being used in attacks (Table 7). A majority of the users also agreed with our choice of method privilege and the three access rights as indicators of damage potential and effort; the findings with respect to channel protocol and data item type are not statistically significant and hence not conclusive (Table 8).

The subjects who disagreed with our choice were of the opinion that a channel's (data item's) damage potential depends on the methods that process the data received from the channel (data item); hence they concluded that a `TCP socket` and an `RPC end point` are equally attractive to an attacker, irrespective of their protocol. These findings suggest that we should assign the same damage potential, i.e., 1, to all channels and data items. In that case, we do not have to perform the difficult step of assigning total orderings among the channel protocols and the data item types.

5.3 Open Source Patch Analysis

We validated our metric's prediction system by establishing a positive correlation between attack surface measurements and software's security risk. A vulnerability patch reduces a system's security risk by removing an exploitable vulnerability from the system; hence we expect the patch to reduce

TABLE 8
The Subjects' Perception about the Attributes

Attribute	Agree	Disagree	Neutral	p-value
Method Privilege	90%	0%	10%	p $<$ 0.0001
Access rights	70%	5%	25%	p = 0.0001
Channel Protocol	45%	25%	30%	p = 0.2967
Access Rights	75%	5%	20%	p $<$ 0.0001
Data Item Type	45%	5%	50%	p = 0.8252
Access Rights	85%	10%	5%	p $<$ 0.0001

the system's attack surface measurement. We demonstrated that a majority of patches in open source software reduce the attack surface measurement.

5.3.1 Identification of Relevant Patches

Not all patches are relevant to the measurement. A patch is relevant if we expect the patch to remove a vulnerability by modifying the number of resources that are part of the attack surface or by modifying such resources' damage potential-effort ratios. For example, we expect a patch that resolves authentication issues to affect the resources' access rights; hence the patch is relevant. We, however, do not always expect the patches for buffer overruns to affect the attack surface measurement.

We used the National Vulnerability Database (NVD) bulletins to decide whether a patch is relevant [31]. The NVD is the US government repository of software vulnerability data. Each bulletin describes a vulnerability and contains a *vulnerability type* assigned, using the Common Weakness Enumeration (CWE) developed at MITRE [32]. The CWE definitions provide information on how to find vulnerabilities in software and how to deal with discovered vulnerabilities. Hence we use a vulnerability's type to decide whether we expect the vulnerability's patch to make the changes mentioned in the previous paragraph, and hence, whether the patch is relevant to the measurement. We identify the following seven types to be relevant: Authentication Issues; Permissions, Privileges, and Access Control; Cross-Site Scripting (XSS); Format String Vulnerability; SQL Injection; OS Command Injection; and Information Disclosure.

Not all relevant patches, however, reduce the attack surface measurement. If a vulnerability has one of the first two types, we always expect the vulnerability's patch to reduce the measurement. The vulnerabilities belonging to the last five types can be patched in different ways; hence the patches may not always reduce the measurement.

5.3.2 Results and Discussion

We analyzed the source code for Firefox and ProFTP vulnerability patches to quantify the change in attack surface measurements. Our results indicate that 67 percent and 70 percent of the relevant patches reduced the attack surfaces of Firefox and the ProFTP server, respectively ($\text{confidence level} = 95\%, \text{p} < 0.05$).

Firefox Results: We analyzed all of the patches released by the Mozilla Foundation for Firefox versions 2.0.0.1 to 2.0.0.8. Mozilla Foundation Security Advisories published 48 `C/C++` vulnerabilities with publicly available patch source codes. Many of the 48 NVD bulletins for the 48 vulnerabilities have no type information. We obtained a data set from MITRE that complements the NVD bulletins' type information [33]. This data set, however, was incomplete; hence we inferred the types from the bulletins' descriptions. We identified 12 out of the 48 patches to be relevant to Firefox's attack surface measurement; eight of these patches reduced the measurement and four did not change the measurement. Three out of the four patches that did not reduce the measurement are patches of three XSS vulnerabilities. We do not expect XSS vulnerability patches to always reduce the attack surface measurement.

ProFTP Results: The ProFTPD project group does not publish any security advisories; hence we searched the NVD to identify 21 ProFTP vulnerabilities with publicly available patch source codes. We inferred the types of vulnerabilities that had missing type information. We identified 10 out of the 21 patches to be relevant to ProFTP's attack surface measurement; seven of these patches reduced the measurement and three format string vulnerability patches did not change the measurement. We do not expect format string vulnerability patches to always reduce the attack surface measurement.

Patches that have a Type Assigned: The missing type inference process is subject to human error. Hence, we repeated our experiment by analyzing only those patches that have a type assigned in the NVD. Our results show that 76.9 percent of the relevant patches reduced the attack surface measurement (confidence $\text{level} = 95\%, \text{p} < 0.05$).

The NVD contains 25,000 bulletins; only 363 bulletins, however, have a vulnerability type and contain one or more hyperlinks to their patches. We identified 73 `C/C++` vulnerability bulletins out of the 363 to be relevant to the attack surface measurement based on their type. We could, however, obtain source code of only 13 patches; in the case of the remaining 60 bulletins, the hyperlinks labeled as patch information point to downloadable patches in binary format (e.g., patches of commercial software). Ten out of these 13 patches reduced the attack surface measurement. One was a format string vulnerability patch and two patches used cryptographic techniques to remove vulnerabilities; hence the three patches did not reduce the measurement.

5.4 Anecdotal Evidence

Anecdotal evidence from industry demonstrates that reducing the attack surface mitigates software security risk. The Sasser worm exploited a buffer overflow vulnerability present in an RPC interface of Windows. The interface was remotely accessible by anyone in Windows 2000 and Windows XP, but was made to be accessible by only local administrators in Windows Server 2003 to reduce the attack surface. The worm could easily spread to Windows 2000 and Windows XP, but not to Windows Server 2003 because of the entry point's higher access rights and hence did not affect Window Server 2003 [34]. Similarly, the Zotob worm and the Nachi worm did not affect some versions of Windows due to reduction in their attack surfaces.

Microsoft uses *Kill-Bits* to block vulnerable `ActiveX controls` and `COM` objects from being hosted in browsers and other scriptable environments [35]. Kill-Bits reduce the browser's attack surface by reducing the amount of running code and make the browser immune to the vulnerabilities present in `ActiveX controls` and `COM` objects.

Both Firefox 2.0 and Firefox 1.5 contained a buffer overflow vulnerability in the SSL 2 protocol implementation. SSL 2 protocol was turned off in Firefox 2.0's default configuration to reduce the attack surface [36]. Hence Firefox 2.0's default configuration was immune to attacks that exploit the vulnerability, whereas Firefox 1.5 was not.

6 Measurement Method for SAP Software Systems

We collaborated with SAP, the world's largest enterprise software company, to apply our method to SAP's enterprise-scale software implemented in `Java` [37]. Our motivation behind the collaboration was three-fold. First,

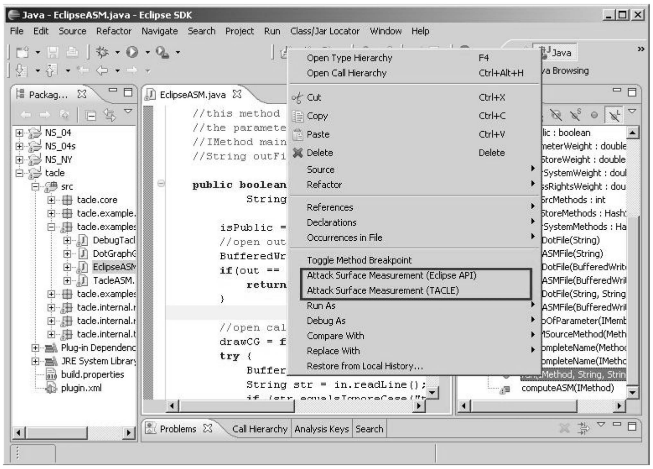

Fig. 9. Attack surface measurement tool implemented as an Eclipse plugin.

we wanted to demonstrate that our method scales to enterprise-scale software and is agnostic to implementation language. Second, we wanted to get SAP developers and architects' feedback on improving our measurement process. Third, we wanted to identify uses of attack surface measurements in multiple phases of the software development process (Section 6.4).

6.1 Implementation of a Measurement Tool

We applied our method to a core component of the SAP NetWeaver platform. The component opens only one TCP channel and uses no persistent data items. Hence we measured the attack surface along the method dimension. We implemented a measurement tool as an Eclipse plugin so that SAP's software developers can use the tool inside their software development environment (Fig. 9) [38].

Similar to C, we identify a system's entry points and exit points from the system's call graph. A method, m, of a system, s, implemented in Java is a direct entry point if either 1) m is in a public *interface* of s and receives data items as input, or 2) m invokes interface methods of a system, s', and receives data items as result, or 3) m invokes a Java I/O library method. Similarly, a method, m, is a direct exit point if either 1) m is in a public interface of s and sends data items as result, or 2) m invokes interface methods of a system, s', and sends data items as input, or 3) m invokes a Java I/O library method.

We use two different techniques to generate the call graph to provide a precision-scalability tradeoff to the software developers: the TACLE Eclipse plugin developed at Ohio State University, which gives a very precise call graph, but does not scale well to large programs [39], and an Eclipse API, which gives a less precise call graph, but scales [40]. The two approaches are complementary: Software developers can use the Eclipse API approach to identify a system's components that are large contributors to the attack surface and then use the TACLE approach on the relevant components to reduce the attack surface.

6.2 Estimation of the Damage Potential-Effort Ratio

The NetWeaver platform's entire codebase runs with only one privilege level. Hence we do not use method privilege to estimate damage potential as we cannot make meaningful suggestions to reduce the attack surface. Instead, we use a method's sources of input data (destinations of output data): an input parameter, the data store, and other systems present in the environment. Different attacks require different sources of input; for example, a method receives data from input parameters in SQL injection attacks and from the data store in File Existence Check attacks. We correlated the three sources of input with possible attacks identified by SAP's internal threat modeling process. For each source, we computed the average severity rating of the attacks that require the source. We assigned numeric values in proportion to the average severity ratings and based on the recommendation of our prior parameter sensitivity analysis (Table 9).

TABLE 9
Numeric Values Assigned to the Sources of Input

Source of Input	Average Severity Rating	Value
other system	1	1
data store	3	18
parameter	5	35

Similar to systems implemented in C, we use a method's access rights to estimate the attacker effort. The NetWeaver platform's public interfaces are accessible to all entities and internal interfaces are accessible only to other NetWeaver components. Hence we identified two access rights levels: public for public interface methods and internal for internal interface methods. We assigned the following numeric values based on our parameter sensitivity analysis' recommendation: public = 1 and internal = 18.

We use the numeric values to compute the numeric damage potential-effort ratios. For example, consider an entry point, m, of a system, s. Suppose m is a public interface method, takes two input parameters, invokes three interfaces methods of another system, and is accessible with the internal access rights level. Then m's damage potential is $2 \times 35 + 3 \times 1 = 73$ and its damage potential-effort ratio is $73/18 = 4.05$.

6.3 Results and Discussion

We measured the attack surfaces of a NetWeaver service's three versions: S1, S2, and S3. The service is a core building block of NetWeaver and is used by most SAP customers. We show the service's entry points and exit points in Table 10 and the measurements in Table 11.

The relative ordering among the service's three measurements conforms to the expected ordering. The S2 version is backward compatible with S1 for customer expectation. Moreover, S2 added new features to S, resulting in an increase in the attack surface measurement. The S3 version

TABLE 10
Entry Points and Exit Points

Version	Count	
	Public	Internal
S3	71	4
S2	67	0
S1	63	0

TABLE 11
Attack Surface Measurements

Version	Attack Surface Measurement
S3	5298.44
S2	4687.00
S1	4649.00

converted a public interface of S2 to an internal interface to mitigate security risk and also added new features to S2. If no new features had been added, S3's measurement would have been less than S2; the increase in S3's measurement is due to new features. The results show that the addition of new features can increase the attack surface measurement. Software developers should, however, aim to minimize the increment. The design decision by SAP developers to introduce internal interfaces was good in that it reduced the overall increase in the measurement.

6.4 Lessons Learned

The measurement results show that our measurement approach is feasible for SAP's complex systems. We also received positive feedback from SAP's developers on the usefulness of the measurement process. They could perform incremental measurements and what-if analysis using our tool. For example, they can easily determine the change in attack surface due to the addition of new features. While reducing the attack surface, they can measure the effects of removing different features; they can also focus on the top contributing methods instead of considering the entire codebase. The tool enables the developers to make informed decisions.

We also learned important lessons on improving our measurement process. For example, our tool guides the developers to focus on a system's relevant parts to reduce the attack surface; the tool, however, does not help in deciding when to stop the reduction process. A possible direction of future work to address this issue is to develop a method to estimate a system's minimum and maximum possible attack surface measurement given the system's functionality.

We also identified several uses of attack surface measurements in the software development process. Attack surface measurements are useful in the design and development phase in an obvious manner, i.e., to reduce risk by reducing the attack surface. We envision four other uses of the measurements. First, in the testing phase, software developers can use the measurements to prioritize the software testing effort. For example, if a system's measurement is high, then they should invest more in testing to identify and remove vulnerabilities; if the measurement is low, they can reduce their effort.

Second, software developers often use manual code inspection to find defects and vulnerabilities in their software [41]. They can use the measurements to prioritize the inspection process, e.g., if a system's measurement is high, then they should spend more resources inspecting the system's code.

Third, in the deployment phase, software consumers can use the measurements to guide their choice of software configuration. Choosing a suitable configuration, especially for complex enterprise-scale software, is a nontrivial and error-prone task. A system's attack surface measurement is dependent on the system's configuration. Hence, assuming that vendors provide attack surface measurements for different configurations of their software, software consumers would choose a configuration that results in a smaller attack surface.

Fourth, in the maintenance phase, software developers can use the measurements as a guide while implementing vulnerability patches. A good patch should not only remove a system's vulnerability, but also should not increase the system's attack surface.

7 Related Work

Our work differs from prior works on quantitative assessment of software security in three key respects. First, our attack surface measurement is based on a system's inherent properties and is independent of the system's vulnerabilities. Previous work assumes the knowledge of a system's vulnerabilities [42], [43], [44], [45], [46], [47]. In contrast, our identification of all entry points and exit points encompasses all known vulnerabilities as well as potential vulnerabilities not yet discovered or exploited. Moreover, a system's attack surface measurement indicates the security risk of the exploitation of the system's vulnerabilities; hence our metric is complementary to and can be used in conjunction with previous work.

Second, prior research on security measurement has taken an *attacker-centric approach* [44], [45], [46], [47]. In contrast, we take a *system-centric approach*. The attacker-centric approach makes assumptions about attacker capabilities and resources, whereas the system-centric approach assesses security without reference to or assumptions about attacker capabilities [48]. Our attack surface measurement is based on a system's design and is independent of the attacker's capabilities and behavior; hence our metric can be used as a tool in the software design and development process.

Third, much prior work on quantification of security are conceptual in nature and have not been applied to real software systems [42], [47], [49], [50], [51]. In contrast, we applied our method to real systems such as FTP servers, IMAP servers, and an SAP software system.

Alves-Foss et al. use the System Vulnerability Index (SVI) —obtained by evaluating factors such as system characteristics, potentially neglectful acts, and potentially malevolent acts—as a measure of a system's vulnerability [42]. They, however, identify only the relevant factors of operating systems; their focus is on operating systems and not individual or generic software applications. Moreover, they assume that they can quantify all of the factors that determine a system's SVI. In contrast, we assume that we can quantify a resource's damage potential and effort.

Littlewood et al. explore the use of probabilistic methods used in traditional reliability analysis in assessing the operational security of a system [49]. In their conceptual framework, they propose using the effort made by an attacker to breach a system as an appropriate measure of the system's security. They, however, do not propose a concrete method to estimate the attacker effort.

Voas et al. propose a relative security metric based on a fault injection technique [43]. They propose a Minimum-Time-To-Intrusion (MTTI) metric based on the predicted period of time before any simulated intrusion can take place. The MTTI value, however, depends on the threat classes simulated and the intrusion classes observed. In contrast, the attack surface metric does not depend on any threat class. Moreover, the MTTI computation assumes knowledge of system vulnerabilities.

Ortalo et al. model a system's known vulnerabilities as a privilege graph [52] and combine assumptions about the attacker's behavior with the privilege graphs to obtain attack state graphs [44]. They analyze the attack state graphs using Markov techniques to estimate the effort an attacker might spend to exploit the vulnerabilities; the estimated effort is a measure of the system's security. Their technique, however, assumes knowledge of the system's vulnerabilities and the attacker's behavior. Moreover, their approach focuses on assessing the operational security of operating systems and not individual software applications.

Schneier uses attack trees to model the different ways in which a system can be attacked [45]. Given an attacker goal, Schneier constructs an attack tree to identify the different ways in which the goal can be satisfied and to determine the cost to the attacker in satisfying the goal. The estimated cost is a measure of the system's security. Construction of an attack tree, however, assumes the knowledge of the following three factors: system vulnerabilities, possible attacker goals, and the attacker behavior.

McQueen et al. use an estimate of a system's expected Time-To-Compromise (TTC) as an indicator of the system's security risk [46]. TTC is the expected time needed by an attacker to gain a privilege level in a system; TTC, however, depends on the system's vulnerabilities and the attacker's skill level.

8 Conclusion and Future Work

There is a pressing need for practical security metrics and measurements today. In this paper, we formalized a pragmatic approach of attack surface measurement; our method is useful to both software industry and software consumers. Howard's measurement method is already used on a regular basis as part of Microsoft's Security Development Lifecycle. Mu Security's Mu-4000 Security Analyzer uses our measurement framework for security analysis [53]. SAP is also planning to use attack surface measurements in their software quality improvement process.

Our work can be extended in three possible directions. First, our attack surface measurement method requires a system's source code. It may not, however, always be feasible to obtain the source code of a system (e.g., commercial software). Moreover, the size of the codebase may be prohibitively large (e.g., the codebase of a Linux distribution). A useful extension of the measurement method would be to define a systematic way to approximate a system's attack surface measurement in the absence of source code. Second, our I/O automata model is not expressive enough to include attacks such as side channel attacks, covert channel attacks, and attacks where one user of a software system can affect other users (e.g., fork bombs). A possible direction of future work would be to extend our formalization of damage potential and attacker effort to include such attacks. Third, another possible direction of future work would be to explore the use of attack surface measurements in "safe" software composition, e.g., we may consider a composition of two systems, A and B, to be safe if the attack surface measurement of the composition is not greater than the sum of A and B's measurements.

We view our work as a first step in the grander challenge of security metrics. We believe that no single security metric or measurement will be able to fulfill our requirements. We certainly need multiple metrics and measurements to quantify different aspects of security. A possible direction of future work would be to establish a framework for combining multiple security measurements.

Acknowledgments

The authors thank Virgil Gligor, Paul Hoffman, Michael Howard, Yuecel Karabulut, Gaurav Kataria, Dilsun Kaynar, Roy Maxion, Miles McQueen, and Michael Reiter for insightful discussions and their feedback at various stages of the research. They also thank the anonymous reviewers for their suggestions and feedback to improve the paper.

References

[1] *Toward a Safer and More Secure Cyberspace,* S.E. Goodman and H.S. Lin, eds. The Nat'l Academics Press, 2007.

[2] Computing Research Assoc. (CRA), "Four Grand Challenges in Trustworthy Computing," http://www.cra.org/reports/trustworthy.computing.pdf, Nov. 2003.

[3] G. McGraw, "From the Ground Up: The DIMACS Software Security Workshop," *IEEE Security and Privacy,* vol. 1, no. 2, pp. 59-66, Mar./Apr. 2003.

[4] R.B. Vaughn, R.R. Henning, and A. Siraj, "Information Assurance Measures and Metrics—State of Practice and Proposed Taxonomy," *Proc. Hawaii Int'l Conf. System Sciences,* 2003.

[5] SAP AG, "SAP—Business Software Solutions Applications and Services," http://www.sap.com, 2009.

[6] M. Howard, "Fending off Future Attacks by Reducing Attack Surface," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.asp, 2003.

[7] M. Howard, J. Pincus, and J. Wing, "Measuring Relative Attack Surfaces," *Proc. Workshop Advanced Developments in Software and Systems Security,* 2003.

[8] P.K. Manadhata and J.M. Wing, "Measuring a System's Attack Surface," Technical Report CMU-CS-04-102, Carnegie Mellon Univ., Jan. 2004.

[9] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI-Quarterly,* vol. 2, no. 3, pp. 219-246, Sept. 1989.

[10] P.K. Manadhata, "An Attack Surface Metric," PhD dissertation, Carnegie Mellon Univ., Dec. 2008.

[11] Y.Y. Haimes, *Risk Modeling, Assessment, and Management.* Wiley, 2004.

[12] N.E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Trans. Software Eng.,* vol. 25, no. 5, pp. 675-689, Sept./Oct. 1999.

[13] R. Gopalakrishna, E. Spafford, and J. Vitek, "Vulnerability Likelihood: A Probabilistic Approach to Software Assurance," Technical Report CERIAS TR 2005-06, Purdue Univ., 2005.

[14] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis, "Characterizing the Security Vulnerability Likelihood of Software Functions," *Proc. Int'l Conf. Software Maintenance,* 2003.

[15] H. Chen, D. Wagner, and D. Dean, "Setuid Demystified," *Proc. 11th USENIX Security Symp.,* pp. 171-190, 2002.

[16] S. Poznyakoff, "GNU Cflow," http://www.gnu.org/software/cflow, 2009.

[17] B. Kitchenham, S.L. Pfleeger, and N. Fenton, "Towards a Framework for Software Measurement Validation," *IEEE Trans. Software Eng.,* vol. 21, no. 12, pp. 929-944, Dec. 1995.

[18] N. Schneidewind, "Methodology for Validating Software Metrics," *IEEE Trans. Software Eng.,* vol. 18, no. 5, pp. 410-422, May 1992.

[19] E. Weyuker, "Evaluating Software Complexity Measures," *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1357-1365, Sept. 1988.

[20] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach,* 1998.

[21] L. Briand, K.E. Emam, and S. Morasca, "Theoretical and Empirical Validation of Software Product Measures," TR ISERN-95-03, Fraunhofer Inst. for Experimental Software Eng., 1995.

[22] *Handbook of Consumer Psychology,* C.P. Haugtvedt, P.M. Herr, and F.R. Kardes, eds. Psychology Press, 2008.

[23] M.Y. Liu and I. Traore, "Properties for Security Measures of Software Products," *Applied Math. and Information Science J.*, vol. 1, no. 2, pp. 129-156, May 2007.

[24] Microsoft Corporation, "Microsoft Security Bulletin Search," http://www.microsoft.com/technet/security/current.aspx, 2010.

[25] W.R. Shadish, T.D. Cook, and D.T. Campbell, *Experimental and Quasi-Experimental Designs for Generalized Causal Inference.* Houghton Mifflin Company, 2001.

[26] *Handbook of Survey Research,* P. Rossi, J. Wright, and A. Anderson, eds. The Academic Press, 1983.

[27] Y. Miyazaki and K. Mori, "Cocomo Evaluation Tailoring," *Proc. Eighth Int'l Conf. Software Eng.*, pp. 292-299, 1985.

[28] C.F. Kemerer, "An Empirical Validation of Software Cost Estimation Models," *Comm. ACM,* vol. 30, no. 5, pp. 416-429, 1987.

[29] M.G. Mendonça and V.R. Basili, "Validation of an Approach for Improving Existing Measurement Frameworks," *IEEE Trans. Software Eng.*, vol. 26, no. 6, pp. 484-499, June 2000.

[30] R. Likert, "A Technique for the Measurement of Attitudes," *Archives of Psychology,* vol. 22, no. 140, pp. 5-55, June 1932.

[31] NIST, "National Vulnerability Database," http://nvd.nist.gov/, 2010.

[32] MITRE, "CWE—Common Weakness Enumeration," http://cwe.mitre.org/, 2010.

[33] S.M. Christey personal communication, 2007.

[34] M. Howard personal communication, 2005.

[35] Microsoft Security Research and Defense, http://blogs.technet.com/srd/archive/2008/02/06/the-kill_2d00_bit-faq _3a00_part-1-of-3.aspx, Feb. 2008.

[36] G. Markham, "Reducing Attack Surface," http://weblogs.mozillazine.org/gerv/archives/2007/02/reducing_attack_surface.html, 2009.

[37] P.K. Manadhata, Y. Karabulut, and J.M. Wing, "Report: Measuring the Attack Surfaces of Enterprise Software," *Proc. Int'l Symp. Eng. Secure Software and Systems,* 2009.

[38] Eclipse, "Eclipse—An Open Development Platform," http://www.eclipse.org/, 2010.

[39] M. Sharp, J. Sawin, and A. Rountev, "Building a Whole-Program Type Analysis in Eclipse," *Proc. Eclipse Technology Exchange Workshop Object-Oriented Programming, Systems, Languages, and Applications,* pp. 6-10, 2005.

[40] Eclipse, "Eclipse Package org.eclipse.jdt.internal.corext.callhierarchy," http://mobius.inria.fr/eclipse-doc/org/eclipse/jdt/internal/corext/call hierarchy/package-summary.html, 2009.

[41] M. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.*, vol. 15, no. 3, pp. 182-211, 1976.

[42] J. Alves-Foss and S. Barbosa, "Assessing Computer Security Vulnerability," *ACM SIGOPS Operating Systems Rev.*, vol. 29, no. 3, pp. 3-13, 1995.

[43] J. Voas, A. Ghosh, G. McGraw, F. Charron, and K. Miller, "Defining an Adaptive Software Security Metric from a Dynamic Software Failure Tolerance Measure," *Proc. Ann. Conf. Computer Assurance,* 1996.

[44] R. Ortalo, Y. Deswarte, and M. Kaâniche, "Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security," *IEEE Trans. Software Eng.*, vol. 25, no. 5, pp. 633-650, Sept./Oct. 1999.

[45] B. Schneier, "Attack Trees: Modeling Security Threats," *Dr. Dobb's J.*, vol. 24, no. 12, pp. 21-29, 1999.

[46] M.A. McQueen, W.F. Boyer, M.A. Flynn, and G.A. Beitel, "Time-to-Compromise Model for Cyber Risk Reduction Estimation," *Proc. ACM Conf. Computer and Comm. Security Workshop Quality of Protection,* Sept. 2005.

[47] D.J. Leversage and E.J. Byres, "Estimating a System's Mean Time-to-Compromise," *IEEE Security and Privacy,* vol. 6, no. 1, pp. 52-60, Jan./Feb. 2008.

[48] D.M. Nicol, "Modeling and Simulation in Security Evaluation," *IEEE Security and Privacy,* vol. 3, no. 5, pp. 71-74, Sept./Oct. 2005.

[49] B. Littlewood, S. Brocklehurst, N. Fenton, P. Mellor, S. Page, D. Wright, J.D.J. McDermid, and D. Gollman, "Towards Operational Measures of Computer Security," *J. Computer Security,* vol. 2, no. 2/3, pp. 211-230, 1993.

[50] B.B. Madan, K. Goseva-Popstojanova, K. Vaidyanathan, and K.S. Trivedi, "Modeling and Quantification of Security Attributes of Software Systems," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 505-514, 2002.

[51] S.E. Schechter, "Computer Security Strength & Risk: A Quantitative Approach," PhD dissertation, Harvard Univ., 2004.

[52] M. Dacier and Y. Deswarte, "Privilege Graph: An Extension to the Typed Access Matrix Model," *Proc. European Symp. Research in Computer Security,* pp. 319-334, 1994.

[53] MuSecurity, "What Is a Security Analyzer," http://www.musecurity.com/solutions/overview/security.html, 2009.

Pratyusa K. Manadhata received the BTech degree in computer science and engineering from IIT Kanpur, India in 2001, and the PhD degree in computer science from Carnegie Mellon University in 2008. He is a researcher at Symantec Research Labs in Culver City, California. His research interests are in computer security and software engineering. He spent two years in the Indian software industry before joining Carnegie Mellon University. He is a member of the IEEE.

Jeannette M. Wing received the SB, SM, and PhD degrees in computer science, all from the Massachusetts Institute of Technology. She is the President's Professor of Computer Science at Carnegie Mellon University. From 2004-2007 she was the head of the Computer Science Department at Carnegie Mellon and currently, on leave from Carnegie Mellon University, she is serving as the assistant director of the Computer and Information Science and Engineering Directorate at the US National Science Foundation. Her research interests are in the foundations of trustworthy computing. She is a fellow of the IEEE.