A Formal Model for A System's Attack Surface

Pratyusa K. Manadhata, Jeannette M. Wing

HP Laboratories

HPL-2011-115

Keyword(s):

attack surface; attack surface metric; io automata; security metrics;

Abstract:

Practical software security metrics and measurements are essential for secure software development. In this chapter, we introduce the measure of a software system's attack surface as an indicator of the system's security. The larger the attack surface, the more insecure the system. We formalize the notion of a system's attack surface using an I/O automata model of the system and introduce an attack surface metric to measure the attack surface in a systematic manner. Our metric is agnostic to a software system's implementation language and is applicable to systems of all sizes. Software developers can use the metric in multiple phases of the software development process to improve software security. Similarly, software consumers can use the metric in their decision making process to compare alternative software.

External Posting Date: August 6, 2011 [Fulltext]   Approved for External Publication
Internal Posting Date: August 6, 2011 [Fulltext]

A Formal Model for a System's Attack Surface

Pratyusa K. Manadhata and Jeannette M. Wing

Abstract Practical software security metrics and measurements are essential for secure software development. In this chapter, we introduce the measure of a software system's *attack surface* as an indicator of the system's security. The larger the attack surface, the more insecure the system. We formalize the notion of a system's attack surface using an I/O automata model of the system and introduce an *attack surface metric* to measure the attack surface in a systematic manner. Our metric is agnostic to a software system's implementation language and is applicable to systems of all sizes. Software developers can use the metric in multiple phases of the software development process to improve software security. Similarly, software consumers can use the metric in their decision making process to compare alternative software.

1 Introduction

Measurement of security, both qualitatively and quantitatively, has been a long standing challenge to the research community and is of practical import to software industry today [7, 28, 22, 29]. There is a growing demand for secure software as we are increasingly depending on software in our day-to-day life. The software industry has responded to the demands by increasing effort for creating "more secure" products and services (e.g., Microsoft's Trustworthy Computing Initiative and SAP's Software LifeCycle Security efforts). How can industry determine whether this effort is paying off and how can consumers determine whether industry's effort has made a difference? We need security metrics and measurements to gauge progress with respect to security; software developers can use metrics to quantify

Pratyusa K. Manadhata
HP Labs, #301, 5 Vaughn Dr, Princeton, NJ, 08540, e-mail: manadhata@cmu.edu

Jeannette M. Wing
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213 e-mail: wing@cs.cmu.edu

the improvement in security from one version of their software to another and software consumers can use metrics to compare alternative software that provide the same functionality.

In this chapter, we formalize the notion of a system's *attack surface* and use the measure of a system's attack surface as an indicator of the system's security. 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 larger the attack surface, the more insecure the system. We also introduce an *attack surface metric* to measure a system's attack surface in a systematic manner.

Our metric does not preclude future use of the attack surface notion to define other security metrics and measurements. In this chapter, we use the attack surface metric in a *relative* manner, i.e., given two systems, we compare their attack surface measurements to indicate whether one is more secure than another with respect to the attack surface metric. Also, we use the attack surface metric to compare only *similar* systems, i.e., different versions of the same system (e.g., different versions of the Windows operating system) or different systems with similar functionality (e.g., different File Transfer Protocol (FTP) servers). We leave other contexts of use for both notions—attack surface and attack surface metric—as future work.

1.1 Motivation

Our attack surface metric is useful to both software developers and software consumers.

Software vendors have traditionally focused on improving code quality to improve software security and quality. The code quality improvement effort aims toward reducing the number of design and coding errors in software. An error causes software to behave differently from the intended behavior as 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 its vulnerabilities. In practice, however, building large and complex software devoid of errors, and hence security vulnerabilities, remains a very 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 security 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 in them and many of those vulnerabilities will be discovered and exploited. They can, however, minimize the risk associated with the exploitation of these vulnerabilities. One way to minimize the risk is by reducing the attack surfaces of their software. A smaller attack surface makes the exploitation of the vulnerabilities harder and lowers the damage of exploitation and hence mitigates the security risk. As shown in Figure 1, the code quality effort and the attack surface reduction approach are complementary; a complete risk mitigation strategy requires a combination of both. Hence software developers can use our metric as a tool in the software development process to reduce their software's attack surfaces.

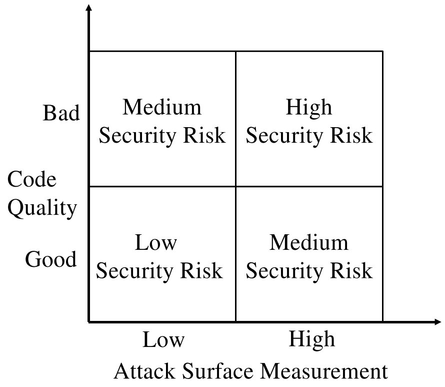

Fig. 1 Attack Surface Reduction and Code Quality Improvement are complementary approaches for mitigating security risk and improving software security.

Software consumers often face the task of choosing one software product from a set of competing and alternative products that provide similar functionality. For example, system administrators often make a choice between different available operating systems, web servers, database servers, and FTP servers for their organization. Several factors such as ease of installation, maintenance, and use, and interoperability with existing enterprise software are relevant to software selection; security, however, is a quality that many consumers care about today and will use in choosing one software system over another. Hence software consumers can use our metric to measure the attack surfaces of alternative software and use the measurements as a guide in their decision making process.

1.2 Attack Surface Metric

We know from the past that many attacks, e.g., exploiting a buffer overflow error, on a system take place by sending data from the system's operating environment into the system. Similarly, many other attacks, e.g., symlink attacks, on a system take place because the system sends data into its environment. In both these types of attacks, an attacker connects to a system using the system's *channels* (e.g., sockets), invokes the system's *methods* (e.g., API), and sends *data items* (e.g., input strings) into the system or receives data items from the system. An attacker can also send data indirectly into a system by using data items that are persistent (e.g., files). An attacker can send data into a system by writing to a file that the system later reads. Similarly, an attacker can receive data indirectly from the system by using shared persistent data items. Hence an attacker uses a system's methods, channels, and data items present in the system's environment to attack the system. We collectively refer to a system's methods, channels, and data items as the system's *resources* and thus define a system's attack surface in terms of the system's resources (Figure 2).

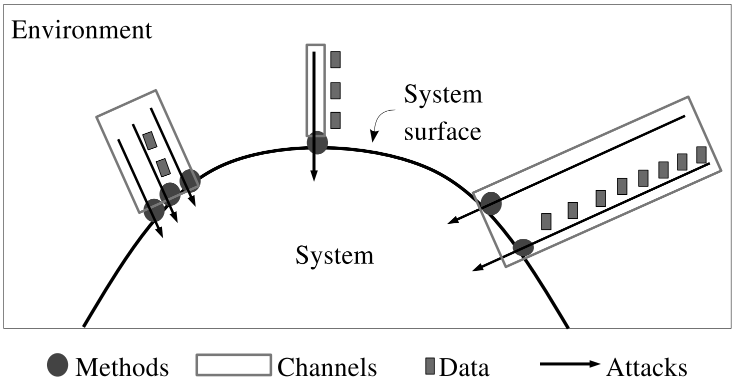

Fig. 2 Intuitively, a system's attack surface is the subset of the system's resources (methods, channels, and data) used in attacks on the system.

Not all resources, however, are part of the attack surface and not all resources contribute equally to the attack surface measurement. In order to measure a system's attack surface, we need to identify the relevant resources that are part of the system's attack surface and to determine the contribution of each such resource to the system's attack surface measurement. A resource is part of the attack surface if an attacker can use the resource in attacks on the system; we introduce an *entry point and exit point framework* to identify these relevant resources. A resource's contribution to the attack surface measurement reflects the likelihood of the resource 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 `non-root` privilege. We introduce the notion of a *damage potential-effort ratio* to estimate a resource's contribution to the attack surface measurement. A system's attack surface measurement is the total contribution of the resources along the methods, channels, and data dimensions; the measurement indicates the level of damage an attacker can potentially cause to the system and the effort required for the attacker to cause such damage. 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.

A system's attack surface measurement does not represent the system's code quality; hence a large attack surface measurement does not imply that the system has many vulnerabilities and having few vulnerabilities in a system does not imply a small attack surface measurement. Instead, a larger attack surface 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. Since a system's code is likely to contain vulnerabilities, it is prudent for software developers to reduce their software's attack surfaces and for software consumers to choose software with smaller attack surfaces to mitigate security risk.

1.3 Roadmap

The rest of this chapter is organized as follows. We briefly discuss the inspiration behind our research in Section 2. In Section 3, we introduce the *entry point and exit point framework* based on the I/O automata model and define a system's attack surface in terms of the framework. In Section 4, we introduce the notions of *damage potential* and *effort* to estimate a resource's contribution to the attack surface; we also define a qualitative measure of the attack surface. We define a *quantitative* measure of the attack surface and introduce an *abstract method* to quantify the attack surface in Section 5. In Section 6, we briefly discuss empirical attack surface measurement results and validation studies. We compare our work with related work in Section 7 and conclude with a discussion of future work in Section 8.

2 Motivation

Our research on attack surface measurement is inspired by Michael Howard's Relative Attack Surface Quotient (RASQ) measurements [11]. We generalized Howard's method and applied the method to four versions of the Linux operating system [20].

2.1 Windows Measurements

Michael Howard of Microsoft informally introduced the notion of attack surface for the Windows operating system and Pincus and Wing further elaborated on Howard's informal notion [13]. The first step in Howard's method is the identification of the *attack vectors* of Windows, i.e., the features of Windows often used in attacks on Windows. Examples of such features are services running on Windows, open sockets, dynamic web pages, and enabled guest accounts. Not all features, however, are equally likely to be used in attacks on Windows. 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 the assignment of weights to the attack vectors to reflect their *attackability*, i.e., the likelihood of a feature being used in attacks on Windows. The weight assigned to an attack vector is the attack vector's contribution to the attack surface. The final step in Howard's method is the estimation of the total 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 total attack surface.

Howard, Pincus, and Wing applied Howard's measurement method to seven versions of the Windows operating system. They identified twenty 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 measurement method was adhoc in nature and was based on intuition; the measurement results, however, confirmed perceived belief about the relative security of the seven versions of Windows. For example, Windows 2000 was perceived to have improved security compared to Windows NT [16]. The measurement results showed that Windows 2000 has a smaller attack surface than Windows NT; hence the measurements reflected the general perception. Similarly, the measurements showed that Windows Server 2003 has the smallest attack surface among the seven versions. The measurement is consistent with observed behavior in several ways, e.g., the relative susceptibility of the versions to worms such as Code Red and Nimda.

2.2 Linux Measurements

We applied Howard's measurement method to four versions of Linux (three RedHat and one Debian) to understand the challenges in applying the method and then to define an improved measurement method.

Howard's method did not have a formal definition of a system's attack vectors. Hence there was no systematic way to identify Linux's attack vectors. We used the history of attacks on Linux to identify Linux's attack vectors. We identified the features of Linux appearing in public vulnerability bulletins such MITRE Common Vulnerability and Exposures (CVE), Computer Emergency Response Team (CERT) Advisories, Debian Security Advisories, and Red Hat Security Advisories; these features are often used in attacks on Linux. We categorized these features into fourteen attack vectors.

Howard, Pincus, and Wing used their intuition and expertise of Windows security to assign weights in the Windows measurements. Their method, however, did not include any suggestions on assigning weights to other software systems' attack vectors. We could not determine a systematic way to assign weights to Linux's attack vectors. Hence we did not assign explicit numeric weights to the attack vectors; we assumed that each attack vector has the same weight. We then counted the number of instances of each attack vector for the four versions of Linux and compared the numbers to get a relative measure of the four versions' attack surfaces.

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 measurement method. First, Howard's method is based on informal notions of a system's attack surface and attack vectors; hence there is no systematic method to identify the attack vectors and to assign weights to them. 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, non-experts 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 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 I/O Automata Model

In this section, we introduce the entry point and exit point framework and use the framework to define a system's attack surface. Informally, a system's *entry points* are the ways through which data "enters" into the system from its environment and *exit points* are the ways through which data "exits" 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 a system's entry points and the exit points act as the basis for attacks on the system.

3.1 I/O Automaton

We model a system and the entities present in its environment as I/O automata [18]. We chose I/O automata as our model for two reasons. First, our notions of entry points and exit points map naturally to an I/O automaton's *input actions* and *output actions*. Second, the I/O automaton's *composition* property allows us to easily reason 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 non-empty 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. In contrast, 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. When we compose a set of automata, we identify different automata's same-named actions; we identify an automaton's output action, m, with the input action m of each automaton having m as an input action. When an automaton having m as an output action performs m, all automata having m as an input action perform m simultaneously. The composition of a set of I/O automata results in an I/O automaton.

3.2 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. The system s interacts with its environment E_s; hence we define the entry points and exit points of s with respect to E_s. Figure 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.

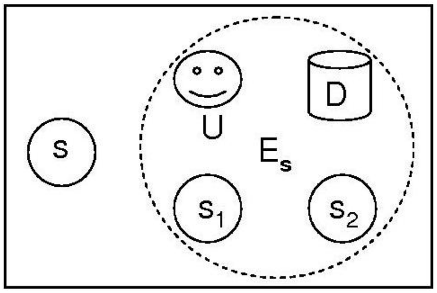

Fig. 3 A system, s, and its environment, E_s.

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 post conditions: for an action, m, $m.pre$ and $m.post$ are the pre and post conditions 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 post conditions 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 has a set of *communication channels*. A system, 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 TCP/UDP sockets and named pipes. We model each channel of a system as a special state variable of the system.

We also model the user U and the data store D as I/O automata. The user U and the data store 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 the data store D as a separate entity to allow sharing of data among the systems in S. The data store D is a set of typed *data items*. Specific examples of data items are strings, URLs, files, and cookies. For every data item, $d \in D$, D has an output action, $read_d$, and an input action, $write_d$. A system, s, or the user U reads d from the data store through the invocation of $read_d$ and writes d to the data store through the invocation of $write_d$. To model global sharing of the data items, corresponding to each data item $d \in D$, we add a state variable, d, to every system, $s \in S$, and the user U. When the system s (or U) reads the data item d from the data store, the value of the data item is written to the state variable d of s (or U). Similarly, when s (or U) writes the data item d to the data store, the value of the state variable d of s (or U) is written to the data item d of the data store.

3.3 Entry Points

The methods in a system's codebase that receive data from the system's environment are the system's entry points. A system's methods can receive data directly or indirectly from the environment. A method, m, of a system, s, receives data items *directly* if either (i.) the user U (Figure 4.a) or a system, s', (Figure 4.b) in the environment invokes m and passes data items as input to m, or (ii.) m reads data items from the data store (Figure 4.c), or (iii.) m invokes a method of a system, s', in the environment and receives data items as results returned (Figure 4.d). A method is a *direct entry point* if it receives data items directly from the environment. Examples of the direct entry points of a web server are the methods in the API of the web server, the methods of the web server that read configuration files, and the methods of the web server that invoke the API of an application server.

In the I/O automata model, a system, s, can receive data from the environment if s has an input action, m, and an entity in the environment has a same-named output action, m. When the entity performs the output action m, s performs the input action m and data is transmitted from the entity to s. We formalize the scenarios when a system, $s' \in T$, invokes m (Figure 4.b) or when m invokes a method of s' (Figure 4.d) 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 the system s is an input action, m, of s such that either (i.) the user U has the output action m (Figure 4.a), or (ii.) a system, $s' \in T$,

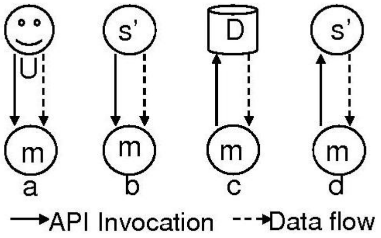

Fig. 4 Direct Entry Point.

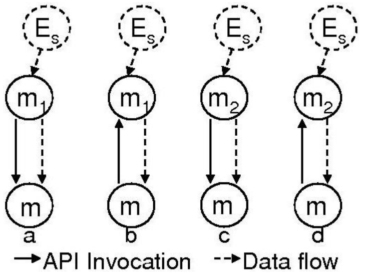

Fig. 5 Indirect Entry Point.

has the output action m (Figure 4.b and Figure 4.d), or (iii.) the data store D has the output action m (Figure 4.c).

A method, m, of s receives data items *indirectly* if either (i.) a method, m_1, of s receives a data item, d, directly, and either m_1 passes d as input to m (Figure 5.a) or m receives d as result returned from m_1 (Figure 5.b), or (ii.) a method, m_2, of s receives a data item, d, indirectly, and either m_2 passes d as input to m (Figure 5.c) or m receives d as result returned from m_2 (Figure 5.d). A method is an *indirect entry point* if it receives data items indirectly from the environment. For example, a method in the API of the web server that receives login information from a user might pass the information to another method in the authentication module; the method in the API is a direct entry point and the method in the authentication module 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 an I/O automaton's internal actions to formalize the system's indirect entry points. We formalize data transmission using the pre and post conditions of a system's actions. If an input action, m, of a system, s, receives a data item, d, directly from the environment, then the subsequent behavior of the system s depends on the value of d; hence d appears in the post condition of m and we write $d \in Res(m.post)$ where $Res : predicate \rightarrow 2^{Var}$ is a function such that for each post condition (or pre condition), p, $Res(p)$ is the set of resources appearing in p. Similarly, if an action, m, of s receives a data item d from another action, m_1, of s, then d appears in the post condition of m_1 and in the pre condition of m. Similar to the direct entry points, we formalize the scenarios Figure 5.a and Figure 5.b the same way and the scenarios Figure 5.c and Figure 5.d the same way. We define indirect entry points recursively.

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

The set of entry points of s is the union of the set of direct entry points and the set of indirect entry points of s.

3.4 Exit Points

A system's methods that send data to the system's environment are the system's exit points. For example, a method that writes into a log file is an exit point. A system's methods can send data directly or indirectly into the environment. A method, m, of a system, s, sends data items *directly* if either (i.) the user U (Figure 6.a) or a system, s', (Figure 6.b) in the environment invokes m and receives data items as results returned from m, or (ii.) m writes data items to the data store (Figure 6.c), or (iii.) m invokes a method of a system, s', in the environment and passes data items as input (Figure 6.d).

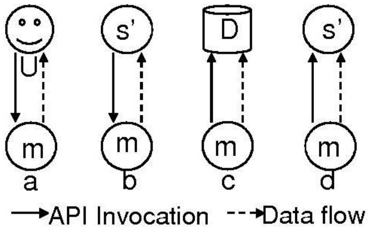

Fig. 6 Direct Exit Point.

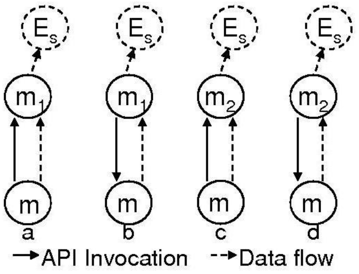

Fig. 7 Indirect Exit Point.

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 in the environment has a same-named input action, m. When s performs the output action m, the entity performs the input action m and data is transmitted from s to the entity.

Definition 3. A *direct exit point* of the system s is an output action, m, of s such that either (i.) the user U has the input action m (Figure 6.a), or (ii.) a system, $s' \in T$, has the input action m (Figure 6.b and Figure 6.d) , or (iii.) the data store D has the input action m (Figure 6.c).

A method, m, of s sends data items *indirectly* to the environment if either (i.) m passes a data item, d, as input to a direct exit point, m_1 (Figure 7.a), or m_1 receives a data item, d, as result returned from m (Figure 7.b), and m_1 sends d directly to the environment, or (ii.) m passes a data item, d, as input to an indirect exit point, m_2 (Figure 7.c), or m_2 receives a data item, d, as result returned from m (Figure 7.d), 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 of a system using an I/O automaton's internal actions. If an output action, m, sends a data item, d, to the environment, then the subsequent behavior of the environment depends on the value of d. Hence d appears in the pre condition of m and in the post condition of the same-named input action m of an entity in the environment. Again we define indirect exit points recursively.

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

The set of exit points of s is the union of the set of direct exit points and the set of indirect exit points of s.

3.5 Channels

An attacker uses a system's channels to connect to the system and invoke a system's methods. Hence a system's 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 a channel, c, of s; hence in our I/O automata model, c appears in the pre condition of a direct entry point (or exit point), m, i.e., $c \in Res(m.pre)$. In our model, every channel of s must appear in the pre condition of at least one direct entry point (or exit point) of s. Similarly, at least one channel must appear in the pre condition of every direct entry point (or direct exit point).

3.6 Untrusted Data Items

The data store D is a collection of persistent and transient data items. The data items that are visible to both a system, s, and the user 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, hence U can use the persistent data items to send (receive) data indirectly into (from) s. For example, s might read a file from the data store after U writes the file to the data store. Hence the persistent data items act as another basis for attacks on s. 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 (i.) $\exists$ a direct entry point, m, of s such that $d \in Res(m.post)$, or (ii.) $\exists$ a direct exit point, m, of s such that $d \in Res(m.pre)$.

Notice that an attacker sends (receives) the transient data items directly into (from) s by invoking s's direct entry points (direct exit points). Since s's direct entry points (direct exit points) act as a basis for attacks on s, we do not consider the transient data items as a different basis for attacks on s. The transient data items are untrusted data items; they are, however, already ”counted” in our definition of direct entry points and direct exit points.

3.7 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 the set of entry points and exit points, C^{E_s} is the set of channels, and I^{E_s} is the set of untrusted data items of s.

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, $E = \langle U, D, T\rangle$, the attack surface, $\langle M_A^E, C_A^E, I_A^E\rangle$, of a system, A, is larger than the attack surface, $\langle M_B^E, C_B^E, I_B^E\rangle$, of a system, B iff either (i.) $M_A^E \supset M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supseteq I_B^E$, or (ii.) $M_A^E \supseteq M_B^E \wedge C_A^E \supset C_B^E \wedge I_A^E \supseteq I_B^E$, or (iii.) $M_A^E \supseteq M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supset I_B^E$.

3.8 Relation between Attack Surface and Potential Attacks

Consider a system, A, and its environment, $E_A = \langle U, D,\ T\rangle$. We model A's interaction with the entities present in its environment as parallel composition, $A||E_A$. Notice that an attacker can send data into A by invoking A's input actions and the attacker can receive data from A when A executes its output actions. Since an attacker attacks a system either by sending data into the system or by receiving data from the system, any schedule of the composition of A and E_A that contains A's input actions or output actions is a potential attack on A. We denote the set of potential attacks on s as $attacks(A)$.

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, $P = s\,||\,U\,||\,D\,||\,(||_{t\in T}\, t)$, such that an input action (or output action), m, of s appears in β.

Note that s's schedules may contain internal actions, but in order for a schedule to be an attack, the schedule must contain at least one input action or output action.

We model an attacker by a set of attacks in our I/O automata model. In other models of security, e.g., for cryptography, an attacker is modeled not just by a set of attacks but also by its power and privilege [6]. Examples of an attacker's power and privilege are the attacker's skill level (e.g., script kiddies, experts, and government agencies) and the attacker's resources (e.g., computing power, storage, and tools). We, however, do not model the attacker's power and privilege in our I/O automata model. Hence our notion of attack surface is independent of the attacker's power and privilege and is dependent only on a system's design and inherent properties.

We show that with respect to the same attacker and operating environment, if a system, A, has a larger attack surface compared to a similar system, B, then the number of potential attacks on A is larger than B. Since A and B are similar systems, we assume both A and B have the same set of state variables and the same sets of resources except the ones appearing in the attack surfaces.

Theorem 1. *Given an environment, $E = \langle U, D, T\rangle$, if the attack surface, $\langle M_A^E, C_A^E, I_A^E\rangle$, of a system, A, is larger than the attack surface, $\langle M_B^E, C_B^E, I_B^E\rangle$, of a system, B, then the rest of the resources of A and B being equal $attacks(A) \supset attacks(B)$.*

Proof. (Sketch)

- Case i: $M_A^E \supset M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supseteq I_B^E$
  Without loss of generality, we assume that $M_A^E \setminus M_B^E = \{m\}$. Consider the compositions $P_A = A\,||\,U\,||\,D\,||\ (||_{t\in T}\, t)$ and $P_B = B\,||\,U\,||\,D\,||\,(||_{t\in T}\, t)$. Any method, $m \in M_B^E$, that is enabled in a state, s_B, of B is also enabled in the corresponding state s_A of A and for any transition, $\langle s_B, m, s'_B\rangle$, of P_B, there is a corresponding transition, $\langle s_A, m, s'_A\rangle$, of P_A. Hence for any schedule $\beta \in attacks(B)$, $\beta \in attacks(A)$ and $attacks(A) \supseteq attacks(B)$.

  - Case a: m is a direct entry point (or exit point) of A.
    Since m is a direct entry point (or exit point), there is an output (or input) action m of either U, D, or a system, $t \in T$. Hence there is at least one schedule, β, of P_A containing m. Moreover, β is not a schedule of P_B as $m \notin M_B^E$. Since β is a potential attack on A, $\beta \in attacks(A) \wedge \beta \notin attacks(B)$. Hence $attacks(A) \supset attacks(B)$.
  - Case b: m is an indirect entry point (or exit point) of A.
    Since m is an indirect entry point (or exit point) of A, there is a direct entry point (or exit point), m_A, of A such that $m_A.post \Rightarrow m.pre$ (or $m.post \Rightarrow m_A.pre$). Hence there is at least one schedule, β, of P_A such that m follows m_A (or m_A follows m) in β. Moreover, β is not an schedule of P_B as $m \notin M_B^E$. Since β is a potential attack on A, $\beta \in attacks(A) \wedge \beta \notin attacks(B)$. Hence $attacks(A) \supset attacks(B)$.

- Case ii: $M_A^E \supseteq M_B^E \wedge C_A^E \supset C_B^E \wedge I_A^E \supseteq I_B^E$
  Without loss of generality, we assume that $C_A^E \backslash C_B^E = \{c\}$. We know that c appears in the pre condition of a direct entry point (or exit point), $m \in M_A^E$. But $c \notin C_B^E$, hence m is never enabled in any state of B and $m \notin M_B^E$. Hence $M_A^E \supset M_B^E$ and from Case i, $attacks(A) \supset attacks(B)$.
- Case iii: $M_A^E \supseteq M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supset I_B^E$
  The proof is similar to case ii.

Theorem 1 has practical significance in the software development process. The theorem shows that if we create a software system's newer version by only adding more resources to an older version, then assuming all resources are counted equally (see Section 4), the newer version has a larger attack surface and hence a larger number of potential attacks. Software developers should ideally strive towards reducing their software's attack surface from one version to another or if adding resources to the software (e.g., adding methods to an API), then do so knowingly that they are increasing the attack surface.

4 Damage Potential and Effort

Not all resources contribute equally to the measure of a system's attack surface 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 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 the damage potential and effort of a resource, r, of a system, s, 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. Examples of attributes are method privilege, access rights, channel protocol, and data item type. Our estimation method is a specific instantiation of our general measurement framework. Our estimation of damage potential 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 the privilege of a method 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 a system's channels to connect to a system and send (receive) data to (from) a 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 the channel's 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` can not. The attacker can send executable code into the system by using a `file` in an attack, but the attacker can not do the same using a `registry entry`. Hence we estimate a data item's damage potential in terms of the data item's 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 the effort the attacker needs to spend to use a resource in an attack in terms of the resource's access rights.

We assume that we have a total ordering, $\succ$, among the values of each of the six attributes, 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 `non-root` privilege; hence `root` $\succ$ `non-root`. 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 to the attack surface than a resource, r_2.

Definition 9. Given two resources, r_1 and r_2, of a system, A, $r_1 \succ r_2$ iff either (i.) $r_1.dp \succ r_2.dp \wedge r_2.ef \succ r_1.ef$, or (ii.) $r_1.dp = r_2.dp \wedge r_2.ef \succ r_1.ef$, or (iii.) $r_1.dp \succ r_2.dp \wedge r_2.ef = r_1.ef$.

Definition 10. Given two resources, r_1 and r_2, of a system, A, $r_1 \succeq r_2$ iff either (i.) $r_1 \succ r_2$ or (ii.) $r_1.dp = r_2.dp \wedge r_2.ef = r_1.ef$.

4.1 Modeling Damage Potential and Effort

In our I/O automata model, we use an action's pre and post conditions to formalize effort and damage potential, respectively. We present a parametric definition of an action, m, of a system, s, below. For simplicity, we assume that 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.

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

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, the parameters MB, CB, and DB represent the benefit to the attacker in using the method m, the channel c, and the data item d in an attack, respectively. R_{pre} is the part of m's pre condition 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 pre condition are analogous. Similarly, R_{post} is the part of m's post condition 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.

We use the total orderings $\succ$ among the values of the attributes to define the notion of weaker (and stronger) pre conditions and post conditions. We first introduce a predicate, $\langle m_1, c_1, d_1 \rangle \succ_{at} \langle m_2, c_2, d_2 \rangle$, to compare the values of an attribute, $at \in \{dp, ef\}$, of the two triples, $\langle m_1, c_1, d_1 \rangle$ and $\langle m_2, c_2, d_2 \rangle$. We later use the predicate to compare pre and post conditions.

Definition 11. Given two methods, m_1 and m_2, two channels, c_1 and c_2, two data items, d_1 and d_2, and an attribute, $at \in \{dp, ef\}$, $\langle m_1, c_1, d_1 \rangle \succ_{at} \langle m_2, c_2, d_2 \rangle$ iff either (i.) $m_1.at \succ m_2.at \wedge c_1.at \succeq c_2.at \wedge d_1.at \succeq d_2.at$, or (ii.) $m_1.at \succeq m_2.at \wedge c_1.at \succ c_2.at \wedge d_1.at \succeq d_2.at$ or (iii.) $m_1.at \succeq m_2.at \wedge c_1.at \succeq c_2.at \wedge d_1.at \succ d_2.at$.

Consider two methods, m_1 and m_2. We say that m_1 has a weaker pre condition than m_2 iff $(m_1.R_{pre} = m_2.R_{pre}) \wedge (m_2.pre \Rightarrow m_1.pre)$. We only compare the parts of the pre conditions involving the access rights and assume that the rest of the pre conditions are the same for both m_1 and m_2. Notice that if m_1 has a lower access rights level than m_2, i.e., $m_2.ef \succ m_1.ef$, then for all access rights levels MA, $(MA \succeq m_2.ef) \Rightarrow (MA \succeq m_1.ef)$; the rest of the clauses in the pre conditions are analogous. Hence we define the notion of weaker pre condition as follows.

Definition 12. Given the pre condition, $m_1.pre = (R_{pre} \wedge MA \succeq m_1.ef \wedge CA \succeq c_1.ef \wedge DA \succeq d_1.ef)$, of a method, m_1, and the pre condition, $m_2.pre = (R_{pre} \wedge MA \succeq m_2.ef \wedge CA \succeq c_2.ef \wedge DA \succeq d_2.ef)$, of a method, m_2, $m_2.pre \Rightarrow m_1.pre$ if $\langle m_2, c_2, d_2 \rangle \succ_{ef} \langle m_1, c_1, d_1 \rangle$.

We say that m_1 has a weaker post condition than m_2 iff $(m_1.R_{post} = m_2.R_{post}) \wedge (m_1.post \Rightarrow m_2.post)$.

Definition 13. Given the post condition, $m_1.post = (R_{post} \wedge MB \succeq m_1.dp \wedge CB \succeq c_1.dp \wedge DB \succeq d_1.dp)$, of a method, m_1 and the post condition, $m_2.post = (R_{post} \wedge MB \succeq m_2.dp \wedge CB \succeq c_2.dp \wedge DB \succeq d_2.dp)$, of a method, m_2, $m_1.post \Rightarrow m_2.post$ if $\langle m_1, c_1, d_1 \rangle \succ_{dp} \langle m_2, c_2, d_2 \rangle$.

4.2 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'a 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 the attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, of a system, A, 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 14. Given an environment, $E = \langle U, D, T \rangle$, the attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, of a system, A, and the attack surface, $\langle M_B^E, C_B^E, I_B^E \rangle$, of a system, B, A has a larger attack surface measurement than B ($A \gg B$) 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, $\mathbb{R}_{\mathbb{AB}} \subseteq R_B$, of resources such that $\forall r \in \mathbb{R}_{\mathbb{AB}} . A.r \succ B.r$ and $\forall r \in (R_B \setminus \mathbb{R}_{\mathbb{AB}}) . A.r = B.r$.

From Definitions 7 and 14, $\gg$ is transitive. For example, given three systems, A, B, and C, if A has a larger attack surface measurement than B and B has a larger attack surface measurement than C, then A has a larger attack surface measurement than C.

Theorem 2. *Given an environment, $E = \langle U, D, T \rangle$, the attack surface, R_A, of a system, A, the attack surface, R_B, of a system, B, and the attack surface, C, of a system, R_C, if $A \gg B$ and $B \gg C$, then $A \gg C$.*

Proof. (Sketch) From Definition 14, A's attack surface measurement can be larger than B's in two different ways. Similarly, B's attack surface measurement can be larger than C's in two different ways. Hence we consider four different cases in proving the theorem.

- Case 1: $R_A \supset R_B$ and $\forall r \in R_B . A.r \succeq B.r$.

  - Case 1.1: $R_B \supset R_C$ and $\forall r \in R_C . B.r \succeq C.r$.
    Since $R_A \supset R_B$ and $R_B \supset R_C$, $R_A \supset R_C$. Also, since $R_B \supset R_C$ and $\forall r \in R_B . A.r \succeq B.r$, $\forall r \in R_C . A.r \succeq B.r$. From the assumptions of Case 1.1, $\forall r \in R_C . B.r \succeq C.r$. Hence $\forall r \in R_C . A.r \succeq B.r \succeq C.r$. Hence $A \gg C$.
  - Case 1.2: $R_B = R_C$ and there is a nonempty set, $\mathbb{R}_{\mathbb{BC}} \subseteq R_C$, of resources such that $\forall r \in \mathbb{R}_{\mathbb{BC}} . B.r \succ C.r$ and $\forall r \in (R_C \setminus \mathbb{R}_{\mathbb{BC}}) . B.r = C.r$.
    Since $R_A \supset R_B$ and $R_B = R_C$, $R_A \supset R_C$. Consider a resource, $r \in R_C$. From the assumptions of Case 1.2, if $r \in \mathbb{R}_{\mathbb{BC}}$, then $B.r \succ C.r$, and if $r \in (R_C \setminus \mathbb{R}_{\mathbb{BC}})$, then $B.r = C.r$. Hence $\forall r \in R_C . B.r \succeq C.r$. Also, from the assumptions of Case 1, $\forall r \in R_B . A.r \succeq B.r$. Since $R_B = R_C$, $\forall r \in R_C . A.r \succeq B.r \succeq C.r$. Hence $A \gg C$.

- Case 2: $R_A = R_B$ and there is a nonempty set, $\mathbb{R}_{\mathbb{AB}} \subseteq R_B$, of resources such that $\forall r \in \mathbb{R}_{\mathbb{AB}} . A.r \succ B.r$ and $\forall r \in (R_B \setminus \mathbb{R}_{\mathbb{AB}}) . A.r = B.r$.

  - Case 2.1: $R_B \supset R_C$ and $\forall r \in R_C . B.r \succeq C.r$.
    The proof is similar to Case 1.2.

- Case 2.2: $R_B = R_C$ and there is a nonempty set, $\mathbb{R}_{\mathbb{BC}} \subseteq R_C$, of resources such that $\forall r \in \mathbb{R}_{\mathbb{BC}}.B.r \succ C.r$ and $\forall r \in (R_C \setminus \mathbb{R}_{\mathbb{BC}}).B.r = C.r$.
  Since $R_A = R_B$ and $R_B = R_C$, $R_A = R_C$. Consider the set, $\mathbb{R}_{\mathbb{AC}} = \mathbb{R}_{\mathbb{AB}} \cup \mathbb{R}_{\mathbb{BC}}$, of resources. We shall prove that $\forall r \in \mathbb{R}_{\mathbb{AC}}.A.r \succ C.r$. Consider a resource, $r \in \mathbb{R}_{\mathbb{AC}}$. If $r \in \mathbb{R}_{\mathbb{AB}} \cap \mathbb{R}_{\mathbb{BC}}$, then $A.r \succ B.r \succ C.r$. If $r \in \mathbb{R}_{\mathbb{AB}} \setminus \mathbb{R}_{\mathbb{BC}}$, then $A.r \succ B.r = C.r$. Similarly, if $r \in \mathbb{R}_{\mathbb{BC}} \setminus \mathbb{R}_{\mathbb{AB}}$, then $A.r = B.r \succ C.r$. Hence $\forall r \in \mathbb{R}_{\mathbb{AC}}.A.r \succ C.r$. Also, from the assumptions of Case 2 and Case 2.2, $\forall r \in R_C \setminus \mathbb{R}_{\mathbb{AC}}.A.r = C.r$. Hence $A \gg C$.

The transitivity of $\gg$ has practical implications for attack surface reduction; while reducing A's attack surface measurement compared to C's, software developers should focus on the set $\mathbb{R}_{\mathbb{AC}}$ of resources instead of either the set $\mathbb{R}_{\mathbb{AB}}$ or the set $\mathbb{R}_{\mathbb{BC}}$.

4.3 Relation Between Attack Surface Measurement and Potential Attacks

We show that with respect to the same attacker and operating environment, if a system, A, has a larger attack surface measurement compared to a system, B, then the number of potential attacks on A is larger than B.

Theorem 3. *Given an environment, $E = \langle U, D, T \rangle$, if the attack surface of a system A is the triple $\langle M_A^E, C_A^E, I_A^E \rangle$, the attack surface of of a system, B, is the triple $\langle M_B^E, C_B^E, I_B^E \rangle$, and A has a larger attack surface measurement than B, then $attacks(A) \supseteq attacks(B)$.*

Proof. (Sketch)

- Case 1: This is a corollary of Theorem 1.
- Case 2: $M_A^E = M_B^E \wedge C_A^E = C_B^E \wedge I_A^E = I_B^E$
  Without loss of generality, we assume that $R = \{r\}$ and $A.r \succ B.r$.

  - Case i: $(B.r).ef \succ (A.r).ef \wedge (A.r).dp \succ (B.r).dp$
    From definitions 12 and 13, there is an action, $m_A \in M_A^E$, that has a weaker precondition and a stronger post condition than the same-named action, $m_B \in M_B^E$, i.e.,
    $$(m_B.pre \Rightarrow m_A.pre) \wedge (m_A.post \Rightarrow m_B.post). \quad (1)$$
    Notice that any schedule of the composition P_B (as defined in the proof sketch of Theorem 1) that does not contain m_B is also a schedule of the composition P_A. Now consider a schedule, β, of P_B that contains m_B and the following sequence of actions that appear in β:..$m_1 m_B m_2$... Hence,
    $$(m_1.post \Rightarrow m_B.pre) \wedge (m_B.post \Rightarrow m_2.pre). \quad (2)$$

From equations (1) and (2), $(m_1.post \Rightarrow m_B.pre \Rightarrow m_A.pre) \wedge$ $(m_A.post \Rightarrow m_B.post \Rightarrow m_2.pre)$. Hence, $(m_1.post \Rightarrow m_A.pre) \wedge (m_A.post \Rightarrow m_2.pre)$. That is, we can replace the occurrences of m_B in β with m_A. Hence β is also a schedule of the composition P_A and $attacks(A) \supseteq attacks(B)$.

- Case ii and Case iii: The proof is similar to Case i.

Theorem 3 also has practical significance in the software development process. The theorem shows that if software developers modify the values of a resource's attributes and hence increase the resource's damage potential and/or decrease the resource's effort in their software's newer version, then all else being the same between the two versions, the newer version's attack surface measurement becomes larger and the number of potential attacks on the software increases.

5 A Quantitative Metric

In the previous section, we introduced a qualitative measure of a system's attack surface (Definition 14). The qualitative measure is an ordinal scale [5]; given two systems, we can only determine if one system has a relatively larger attack surface measurement than another. We, however, can not quantify the difference in the measurements.

We need a quantitative measure of the attack surface to quantify the difference in the attack surface measurements. We can also measure the absolute attack surface using the quantitative measure. In this section, we introduce a quantitative measure of the attack surface; the measure is a ratio scale. We quantify a resource's contribution to the attack surface in terms of a *damage potential-effort ratio*.

5.1 Damage Potential-Effort Ratio

In the previous section, in estimating a resource's contribution to the attack surface, we consider the resource's damage potential and effort in isolation. 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 that then enables the attacker to cause damage as well as gain more access rights. Hence we consider a resource's damage potential and effort in tandem and quantify a resource's contribution to the attack surface as a damage potential-effort ratio. The damage potential-effort 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.

We assume a function, der_m: 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: channel $\rightarrow \mathbb{Q}$, for the channels and a function, der_d: 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. For example, we compute a method's damage potential-effort ratio from the numeric values assigned to the method's privilege and access rights. We assign the numeric values according to the total orderings imposed on the attributes and based on our knowledge of a system and its environment. For example, we assume a method running as `root` has a higher damage potential than a method running as `non-root` user; hence `root` > `non-root` user in the total ordering and we assign a higher number to `root` than `non-root` 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 [20].

In terms of our formal I/O automata model, a method, *m*'s, damage potential determines how strong *m*'s post condition is. *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 the pre condition of a method, *m*, then the damage potential-effort ratio of *c* (or *d*) 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.

5.2 Quantitative Attack Surface Measurement Method

We quantify a system's attack surface measurement along three dimensions: methods, channels, and data. We estimate the total contribution of the methods, the total contribution of the channels, and the total contribution of the data items to the attack surface.

Definition 15. Given the attack surface, $\langle M^{E_s}, C^{E_s}, I^{E_s} \rangle$, of a system, *s*, *s*'s attack surface measurement is the triple $\langle \sum_{m \in M^{E_s}} der_m(m), \sum_{c \in C^{E_s}} der_c(c), \sum_{d \in I^{E_s}} der_d(d) \rangle$.

We quantitatively measure a system's attack surface in the following three steps.

1. Given a system, s, and its environment, E_s, we identify a set, M^{E_s}, of entry points and exit points, a set, C^{E_s}, of channels, and a set, I^{E_s}, of untrusted data items of s.
2. We estimate the damage potential-effort ratio, $der_m(m)$, of each method $m \in M^{E_s}$, the damage potential-effort ratio, $der_c(c)$, of each channel $c \in C^{E_s}$, and the damage potential-effort ratio, $der_d(d)$, of each data item $d \in I^{E_s}$.
3. The measure of s's attack surface is $\langle \sum\limits_{m \in M^{E_s}} der_m(m), \sum\limits_{c \in C^{E_s}} der_c(c), \sum\limits_{d \in I^{E_s}} der_d(d) \rangle$.

Our measurement method is analogous to the risk estimation method used in risk modeling [9]. A system's attack surface measurement is an indication of the system's risk from attacks on the system. 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, probability of occurrence is $p(e)$ and consequence is $C(e)$. The events in risk modeling are analogous to a system's resources in our measurement method. The probability of occurrence of an event is analogous to the probability of a successful attack on the system using a resource; if the attack is not successful, then the attacker does not benefit from the attack. 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 a method, 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. The consequence of an event 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, $\langle \sum\limits_{m \in M^{E_s}} p(m) der_m(m), \sum\limits_{c \in C^{E_s}} p(c) der_c(c), \sum\limits_{d \in I^{E_s}} p(d)\ der_d(d) \rangle$, which is also the measure of s's attack surface.

In practice, however, it is difficult to predict defects in software [4] and to estimate the likelihood of vulnerabilities in software [8]. Hence we take a conservative approach in our attack surface measurement method and assume that $p(m) = 1$ for all methods, i.e., every method has an exploitable vulnerability. We assume that 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, the measure of s's attack surface is the triple $\langle \sum\limits_{m \in M^{E_s}} der_m(m), \sum\limits_{c \in C^{E_s}} der_c(c), \sum\limits_{d \in I^{E_s}} der_d(d) \rangle$.

Given two similar systems, A and B, we compare their attack surface measurements along each of the three dimensions to determine if one system is more secure than another along that dimension. There is, however, a seeming contradiction in our measurement method with our intuitive notion of security. For example, consider a system, A, that has 1000 entry points each with a damage potential-effort ratio of 1 and a system, B, that has only one entry point with a damage potential-effort ratio of 999. A has a larger attack surface measurement whereas A is intuitively more secure. This contradiction is due to the presence of *extreme events*, i.e., events that have a significantly higher consequence compared to other events [9]. An entry point with a damage potential-effort ratio of 999 is analogous to an extreme event. In the presence of extreme events, the shortcomings of the risk estimation method used in the previous paragraph is well understood and the partitioned multiobjective risk method is recommended [2]. In our attack surface measurement method, however, we compare the attack surface measurements of similar systems, i.e., systems with comparable sets of resources and comparable damage potential-effort ratios of the resources; hence we do not expect extreme events such as the example shown to arise in practice.

6 Empirical Results

In this section, we briefly discuss our empirical attack surface measurements and exploratory validation studies. Our discussion focuses on the reasons behind each study; please see Manadhata and Wing for details about the studies [21].

6.1 Attack Surface Measurement Results

We introduced an abstract attack surface measurement method in the previous section. We instantiated the method for software implemented in the `C` programming language and demonstrated that our method is applicable to real world software. We measured the attack surfaces of two open source IMAP servers: Courier-IMAP 4.0.1 and Cyrus 2.2.10; we chose the IMAP servers due to their popularity. We considered only the code specific to the IMAP daemon in our measurements to obtain a fair comparison. The Courier and the Cyrus code bases contain nearly 33K and 34K lines of code specific to the IMAP daemon, respectively. We also measured the attack surfaces of two open source FTP daemons: ProFTPD 1.2.10 and Wu-FTPD 2.6.2. The ProFTP codebase contains 28K lines of `C` code and the Wu-FTP codebase contains 26K lines of `C` code. The measurement results conformed to our intuition. For example, the ProFTP project grew out of the Wu-FTP project and was designed and implemented from the ground up to be a more secure and configurable FTP server. Our measurements showed that ProFTPD is more secure than Wu-FTPD along the method dimension.

6.2 Validation Studies

A key challenge in security metrics research is the validation of a metric. Validating a software attribute's measure is hard in general [14]; security is a software 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 software engineering research community's software metrics validation approaches [5].

In practice, validation approaches are based on distinguishing *measures* from *prediction systems*; measures are used to numerically characterize software attributes whereas prediction systems are used to predict software attributes' values. For example, lines of code (LOC) is a measure of software "length;" the measure becomes a prediction system if we use LOC to predict software "complexity." A software measure is validated by establishing that the measure is a proper numerical characterization of an attribute. Similarly, prediction systems are validated by establishing their accuracy via empirical means.

Our attack surface metric plays a dual role: the metric is a measure of a software attribute, i.e., the attack surface and also a prediction system to indicate the security risk of software. Hence we took a two-step approach for validation. First, we validated the measure by validating our attack surface measurement method. Second, we validated the prediction system by validating attack surface measurement results.

We conducted two empirical studies to validate our measurement method: a statistical analysis of data collected from Microsoft Security Bulletins and an expert user survey. Our approach is motivated by the notion of *convergent evidence* in Psychology [10]; 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. Also, the statistical analysis is with respect to Microsoft Windows whereas the expert survey is with respect to Linux. Hence our validation approach is agnostic to operating system and system software.

We validated our metric's prediction system by establishing a positive correlation between attack surface measurements and software's security risk. First, we formally showed that a larger attack surface leads to a larger number of potential attacks on software in the I/O automata model (Section 3.8 and Section 4.3). Second, we established a relationship between attack surface measurements and security risk by analyzing vulnerability patches in open source software. A vulnerability patch reduces a system's security risk by removing an exploitable vulnerability from the system; hence we expect the patch to reduce the system's attack surface measurement. We demonstrated that a majority of patches in open source software, e.g., Firefox and ProFTP server, reduce the attack surface measurement. Third, we gathered anecdotal evidence from software industry to show that attack surface reduction mitigates security risk; for example, the Sasser worm, the Zotob worm, and the Nachi worm did not affect some versions of Windows due to reduction in their attack surfaces [12].

6.3 SAP Software Systems

Our `C` measurements focused on software that are small in their code size and simple in their architectural design. We collaborated with SAP, the world's largest enterprise software company, to apply our method to SAP's enterprise-scale software implemented in `Java`. Our motivation behind the collaboration was two-fold. First, we wanted to demonstrate that our method scales to enterprise-scale software and is agnostic to implementation language. Second, we had the opportunity to interact closely with SAP's software developers and architects and get their feedback on improving our measurement method.

We instantiated our abstract measurement method for the `Java` programming language and implemented a tool to measure the attack surfaces of SAP software implemented in `Java`. We applied our method to three versions of a core SAP component. The measurement results conformed to the three versions' perceived relative security. We also identified multiple uses of attack surface measurements in the software development process. For example, attack surface measurements are useful in the design and development phase to mitigate security risk, in the testing and code inspection phase to guide manual effort, in the deployment phase to choose a secure configuration, and in the maintenance phase to guide vulnerability patch implementation.

7 Related Work

Our attack surface metric differs from prior work in three key aspects. First, our attack surface measurement is based on a system's inherent properties and is independent of any vulnerabilities present in the system. Previous work assumes the knowledge of the known vulnerabilities present in the system [1, 30, 25, 27, 23, 15]. 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 measurement of security has taken an *attacker-centric approach* [25, 27, 23, 15]. 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 a system's security without reference to or assumptions about attacker capabilities [24]. 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, many of the prior works on quantification of security are conceptual in nature and haven't been applied to real software systems [1, 17, 15, 19, 26]. In contrast, we demonstrate the applicability of our metric to real systems by measuring the attack surfaces of two FTP servers, two IMAP servers, and three versions of 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 [1]. 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 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 [17]. In their conceptual framework, they propose to use 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 [30]. 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 the knowledge of system vulnerabilities.

Ortalo et al. model a system's known vulnerabilities as a privilege graph [3] and combine assumptions about the attacker's behavior with the privilege graphs to obtain attack state graphs [25]. 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 the 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 [27]. 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 [23]. 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 Summary and Future Work

There is a pressing need for practical security metrics and measurements today. In this chapter, we formalized the notion of a system's attack surface and introduced a systematic method to measure it. Our pragmatic attack surface measurement approach is useful to both software developers and software consumers.

Our formal model can be extended in two directions. First, we do not make any assumptions about an attacker's resources, capabilities, and behavior in our I/O automata model. In terms of an attacker profile used in cryptography, we do not characterize an attacker's power and privilege. A useful extension of our work would be to include an attacker's power and privilege in our formal I/O automata model.

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). We could extend the current formal model by extending our formalization of damage potential and attacker effort to include such attacks.

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. We also believe that our understanding over time would lead us to more meaningful and useful quantitative security metrics.

References

1. Alves-Foss, J., Barbosa, S.: Assessing computer security vulnerability. ACM SIGOPS Operating Systems Review **29**(3) (1995)
2. Asbeck, E., Haimes, Y.Y.: The partitioned multiobjective risk method. Large Scale Systems **6**(1), 13–38 (1984)
3. Dacier, M., Deswarte, Y.: Privilege graph: An extension to the typed access matrix model. In: Proc. of European Symposium on Research in Computer Security (1994)
4. Fenton, N.E., Neil, M.: A critique of software defect prediction models. IEEE Transactions on Software Engineering **25**(5) (1999)
5. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. PWS Publishing Co., Boston, MA, USA (1998)
6. Gligor, V.D.: Personal communication (2008)
7. Goodman, S.E., Lin, H.S. (eds.): Toward a Safer and More Secure Cyberspace. The National Academics Press (2007)
8. Gopalakrishna, R., Spafford, E., , Vitek, J.: Vulnerability likelihood: A probabilistic approach to software assurance. Tech. Rep. 2005-06, CERIAS, Purdue Univeristy (2005)
9. Haimes, Y.Y.: Risk Modeling, Assessment, and Management. Wiley (2004)
10. Haugtvedt, C.P., Herr, P.M., Kardes, F.R. (eds.): Handbook of Consumer Psychology. Psychology Press (2008)
11. Howard, M.: Fending off future attacks by reducing attack surface. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.asp (2003)
12. Howard, M.: Personal communication (2005)

13. Howard, M., Pincus, J., Wing, J.: Measuring relative attack surfaces. In: Proc. of Workshop on Advanced Developments in Software and Systems Security (2003)
14. Kitchenham, B., Pfleeger, S.L., Fenton, N.: Towards a framework for software measurement validation. IEEE Transactions on Software Engineering **21**(12), 929–944 (1995). DOI http://dx.doi.org/10.1109/32.489070
15. Leversage, D.J., Byres, E.J.: Estimating a system's mean time-to-compromise. IEEE Security and Privacy **6**(1) (2008). DOI http://doi.ieeecomputersociety.org/10.1109/MSP.2008.9
16. Levitt, J.: Windows 2000 security represents a quantum leap. http://www.informationweek.com/834/winsec.htm (2001)
17. Littlewood, B., Brocklehurst, S., Fenton, N., Mellor, P., Page, S., Wright, D., McDermid, J.D.J., Gollman, D.: Towards operational measures of computer security. Journal of Computer Security **2**(2/3), 211–230 (1993)
18. Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI-Quarterly **2**(3) (1989)
19. Madan, B.B., Goseva-Popstojanova, K., Vaidyanathan, K., Trivedi, K.S.: Modeling and quantification of security attributes of software systems. In: DSN, pp. 505–514 (2002)
20. Manadhata, P.K.: An attack surface metric. Ph.D. thesis, Carnegie Mellon University (2008)
21. Manadhata, P.K., Wing, J.M.: An attack surface metric. IEEE Transactions on Software Engineering **99**(PrePrints) (2010). DOI http://doi.ieeecomputersociety.org/10.1109/TSE.2010.60
22. McGraw, G.: From the ground up: The DIMACS software security workshop. IEEE Security and Privacy **1**(2), 59–66 (2003)
23. McQueen, M.A., Boyer, W.F., Flynn, M.A., Beitel, G.A.: Time-to-compromise model for cyber risk reduction estimation. In: ACM CCS Workshop on Quality of Protection (2005)
24. Nicol, D.M.: Modeling and simulation in security evaluation. IEEE Security and Privacy **3**(5), 71–74 (2005)
25. Ortalo, R., Deswarte, Y., Kaâniche, M.: Experimenting with quantitative evaluation tools for monitoring operational security. IEEE Transactions on Software Engineering **25**(5) (1999)
26. Schechter, S.E.: Computer security strength & risk: A quantitative approach. Ph.D. thesis, Harvard University (2004)
27. Schneier, B.: Attack trees: Modeling security threats. Dr. Dobb's Journal (1999)
28. Smith, S.W., Spafford, E.H.: Grand challenges in information security: Process and output. IEEE Security and Privacy **2**, 69–71 (2004). DOI http://doi.ieeecomputersociety.org/10.1109/MSECP.2004.1264859
29. Vaughn, R.B., Henning, R.R., Siraj, A.: Information assurance measures and metrics - state of practice and proposed taxonomy. In: Proc. of Hawaii International Conference on System Sciences (2003)
30. Voas, J., Ghosh, A., McGraw, G., Charron, F., Miller, K.: Defining an adaptive software security metric from a dynamic software failure tolerance measure. In: Proc. of Annual Conference on Computer Assurance (1996)