Designing for Attack Surfaces: Keep Your Friends Close, but Your Enemies Closer

Trent Jaeger[1], Xinyang Ge[1], Divya Muthukumaran[2], Sandra Rueda[3], Joshua Schiffman[4], and Hayawardh Vijayakumar[5]

[1] The Pennsylvania State University, University Park, PA, USA
[2] Imperial College, London, UK
[3] Universidad de Los Andes, Bogota, Colombia
[4] Hewlett-Packard Labs, Bristol, UK
[5] Samsung Research America, Mountain View, CA, USA

Abstract. It is no surprise to say that attackers have the upper hand on security practitioners today when it comes to host security. There are several causes for this problem ranging from unsafe programming languages to the complexity of modern systems at large, but fundamentally, all of the parties involved in constructing and deploying systems lack a methodology for reasoning about the security impact of their design decisions. Previous position papers have focused on identifying particular parties as being "enemies" of security (e.g., users and application developers), and proposed removing their ability to make security-relevant decisions. In this position paper, we take this approach a step further by "keeping the enemies closer," whereby the security ramifications of design and deployment decisions of all parties must be evaluated to determine if they violate security requirements or are inconsistent with other party's assumptions. We propose a methodology whereby application developers, OS distributors, and system administrators propose, evaluate, repair, and test their artifacts to provide a defensible *attack surface*, the set of entry points available to an attacker. We propose the use of a *hierarchical state machine* (HSM) model as a foundation for automatically evaluating attack surfaces for programs, OS access control policies, and network policies. We examine how the methodology tasks can be expressed as problems in the HSM model for each artifact, motivating the possibility of a comprehensive, coherent, and mostly-automated methodology for deploying systems to manage accessibility to attackers.

1 Introduction

It is no surprise to say that attackers have the upper hand on security practitioners today when it comes to host security. For the most part, security practitioners have little insight into where the next exploitable vulnerability will be found, so there seems to be little that they can do to detect and remove vulnerabilities before attackers. For example, a significant effort has been put into reengineering of network-facing servers (e.g., OpenSSH [45] and Postfix mail server [65]), but while these improvements have prevented a variety of new exploits against those daemons, there are so many programs that have access to network data that security practitioners are overwhelmed. On the positive side, many of these programs are run in unprivileged processes, so their compromise does not directly impact the system integrity. However, the negative side is that we (the security community) are even less effective at preventing local exploits than remote exploits.

Finding the root causes of such problems has been difficult. It appears that each party in the construction and deployment of a system is at fault for multiple poor decisions. Application developers clearly are not developing secure code. They still fail to prevent the same types of basic vulnerabilities (e.g., buffer overflows) that we have recognized for years. Further, when given type-safe languages with well-defined formal semantics, developers choose C and various scripting languages, which have all proven very difficult to use securely. OS distributors package the applications together into distributions, but historically, they abdicate responsibility for securing deployment of their distributions to application developers (it's the programs that have the bugs) or the system administrators (they cannot configure systems properly). Despite the introduction of comprehensive mandatory access control (MAC) systems [43, 63] in some distributions, we are still suffering from a variety of local exploits[6]. Finally, the system administrators are left to try to deploy a secure system from insecure parts. It is an impossible task of immense complexity. Currently, in order to deploy a system securely, a system administrator must understand the manner in which attackers can access their systems via the network (which they do pretty well), how the access control policy manages attackers' access to process (such policies are complex), and how programs handle untrusted input data (there are too many interfaces).

A variety of valuable security mechanisms have been developed, but these have not resulted in shifting the balance from the attackers to the defenders. For example, we have known about buffer overflows for many years, so a variety of mechanisms ranging from buffer overflow prevention [14, 44] to protecting the programs execution integrity [1, 10] to controlling the operating sequences that may be invoked [17, 29]. The expense of many of these services has prevented or delayed their adoption, so others have focused on bug detection and prevention (e.g., [5, 40, 46, 68]). Researchers have also developed models that enable reasoning about the integrity of systems [7, 12]. However, the assumptions underlying these models have been in conflict with the practice of deploying systems, and the security community has made little or no headway in changing such practice. Researchers again have developed other models that approximate classical integrity for conventional systems [28, 53, 57], but these require more effort that has not been forthcoming. Finally, system administrators are left to contemplate all the options and trade-offs without any coherent approach to reason about such options. Their task is far too complex.

Returning to the question of who is at fault, it appears that everyone is at one level or another, so the question is how to proceed forward. A number of approaches have been proposed to remove security decision-making from various parties. We agree that the user cannot be trusted to make anything but win-win security decisions [60], but what should be done about the application developers, OS distributors, and system administrators? All need to make decisions that may impact security, but as with users, they also

[6] Part of the problem is that to reduce the complexity of use, these systems are used incompletely, only to protect system services against network attackers, as proposed by other incomplete methods [36, 41].

should only decide among win-win choices with respect to security. Wurster and van Oorschot argue this point for application developers [67], but we argue that this applies to all parties. The challenge is how to apply this approach to all parties. Solworth argues that this will require a fundamental change in systems and programming practice [56], and while we agree that some changes should be encouraged wholeheartedly, we argue that the revolutionary change must be accompanied by a tool-driven methodology that enforces any new requirements comprehensively and usably for each party. Building on the prior analogies, we refer to this approach as "keeping your friends close, but your enemies closer."[7] The idea is that the application developers, OS distributors, and system administrators must work by a methodology that supports decision-making among secure choices rather than giving insecure choices.

In this paper, we propose a methodology for constructing and deploying systems based on the concept of a *hierarchical state machine* [2] (HSM), a model used previously in software model checking. We find that application developers, OS distributors, and system administrators each make decisions that impact security and that these decisions build on one another, requiring a representation that enables checking of conflicts between different party's decisions. The HSM model represents a hierarchy of components (originally, code modules) and their interactions (calls and returns, and resultant data flows), but we find that this approach can be generalized to represent not only program modules, but the data flows that result from the combination of programs into an OS distribution with its access control policy and the data flows that result from the combination of OS distributions into systems of OS distribution instances (which we will call *hosts*, regardless of whether they run on physical or virtual machines) with its network policy (and optionally, virtual machine monitor policy). With the system's data flows expressed in an HSM, the question then is whether we can automate key decisions that these parties make. Our methodology identifies the types of decisions that must be made by each party, which turn out to be the same decisions on different artifacts, and we examine the possibility of automating or at least providing significant automated support for these decisions. While it is early, we are optimistic that such a view of system-wide management of data flow has potential as a new paradigm for achieving secure construction and deployment in practice.

The remainder of the paper is as follows. In Section 2, we examine problems with the current approach to configuring systems and discuss some trends that motivate our proposed approach. In Section 3, we define the concept of attack surfaces, which serves as the basis for decision-making for each party. In Section 4, we outline the proposed approach, showing that application developers, OS distributors, and system administrators need to make decisions that are test against security requirements and be able to build on each others' work effectively. In Section 5, we define our approach, based on the hierarchical state machine model [2] and examine the problems that need to be solved at each methodology task in detail to identify the opportunities and challenges in providing automated support for such tasks.

[7] There seems to be a lot of confusion about the origin of this quote, ranging from Sun-Tzu (Art of War), Machiavelli, and Petrarch, but we were not able to find a definitive source that predates its use in the movie, "The Godfather, Part II."

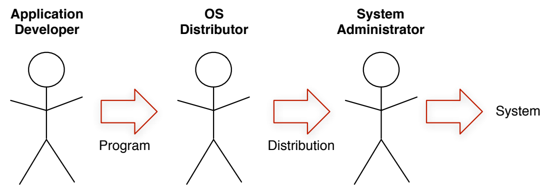

Fig. 1: Application developers, OS distributors, and system administrators are the main parties in constructing software components. The process of constructing a computing system involves the composition of programs into OS distributions and then into systems.

2 Background

In this section, we review how systems are currently configured, the recent trends that may change this situation (hopefully for the better).

2.1 System Configuration

Figure 1 shows the process of configuring a system in terms of the major parties. There would be no system to configure without programs, and application developers provide programs. A *program* consists of one or more executables and scripts, optional program-specific libraries, and deployment-independent data. Operating system (OS) distributors configure one or more programs for their system, including the definition of program packages for installation (including configurations) and security policies. Here, we focus mainly on the access control policy covering the program. System administrators compose systems from one or more OS distributions at a time. These distributions may run on one physical platform or more, thanks to ubiquitous virtualization. System administrators configure network policies to determine how the systems interact, may apply system hardening [6] to improve the security of the system beyond that of the OS distributor, and may change the configurations and access control policies over programs. Users are not shown in Figure 1, as, like many others, we assume that users do not make security-critical decisions [60].

Application developers have a tremendous challenge in building programs that protect themselves from attackers. Nearly any interesting program consists of multiple components, written in multiple languages, by many application developers. Also, software engineering practice has made design approaches that reuse components successful, but many bugs result from incorrect reuse of components. Further, the trade-offs that application developers make often involve compromises of security (e.g., features vs. security) that result in further vulnerabilities. The current state of application developer practice with respect to security is so dismal that Wurster and van Oorschot propose to take security-relevant programming decisions away from application developers through enforcement of best practices [67].

Operating system distributors have an even more difficult challenge in configuring their OS distributions in a manner that ensures security. Historically, the task of OS distributors is to provide an ecosystem for deploying applications easily, flexibly, and with good performance characteristics. While operating systems fundamentally provide protection mechanisms (e.g., address spaces and access control mechanisms), the aim was mainly to keep one program's failure from affecting another program's execution. As a result, security decisions were largely left in the hands of application developers in conventional systems. This approach is inherently incapable of enforcing security [31]. Some operating systems over the years deploy mandatory access control (MAC) for controlling, even malicious, programs, but mandatory access control systems that aim to enforce strong integrity guarantees [4,12,23,30,53] have not seen broad use, and the application of MAC enforcement to conventional systems [41,43] has been hampered by complexity and enforcement of informal goals, such as least privilege [51]. Solworth argues for improved testing effectiveness and reduced complexity in operating systems [56], which we agree are insufficient in current MAC systems.

System administrators are left with the task of configuring the deployment of these OS distributions consisting of many such programs. System administration may consist of many tasks. First, system administrators specify the network policy, which determines how the deployed systems communicate among one another and the Internet at large. Second, they may determine the programs that are run on each system. Third, they may configure various system services based on the site's security requirements. Finally, system administrators are often responsible for the access control policies on their systems. Given that they have many physical machines to manage and they must respond to the non-deterministic behavior of their user community (and, of course, attackers), the security community's assumption that they can perform all of these tasks effectively on such complex systems is misplaced.

Thus, we find that none of these parties is capable of performing the tasks necessary to configure secure systems. Application developers do not ensure that their programs can defend themselves from attackers. OS distributors piece these programs together into systems without understanding the limitations that are built into programs nor configuring systems in a way that ensures any meaningful security property. Finally, system administrators are left with the responsibility to make all of this work. The semantic gap between the fine-grained security decisions in programs and those at the system-level make it impractical for even the best system administrators to configure a secure system unless they terminate all connections to attackers.

2.2 Trends in System Configuration

Despite our current situation, there are some trends that indicate that the kind of revolutionary change needed to develop systems that protect their integrity may be possible.

System Administrators Can Manage Firewalls First, we start with a low-hanging fruit. It appears that network firewalls are an effective approach for protecting systems from attackers. We surmise that firewalls are effective for two reasons: (1) they actually are capable of reducing the accessibility of systems to attackers and (2) system administrators can define firewall policies with little knowledge about program or OS behavior. First, a firewall defines the first line of defense to a system, so its effectiveness is largely independent of how the system is configured behind the firewall. Thus, a firewall rule that denies an attacker access to a particular host prevents the host from being directly accessible to an attacker, regardless of how poorly the OS or programs are built Second, system administrators only need to understand the binding between ports and programs to configure a firewall. This information is standardized, so it is well-known. The result is that system administrators can do an effective job of defining firewall rules, even though these rulebases can get complex [66].

OS Distributors Define MAC Policies MAC policy design unfortunately interacts more subtly with the system's programs, but recent trends in MAC policy configuration involve OS distributors learning more about their programs. MAC policies for SELinux are defined by a small group of experts in SELinux who define policies per program, implying that they study the permissions necessary for the program to run securely. This marks a major shift from OSes supporting arbitrary programs and their security requirements to OS distributors planning for the programs that their OS will run (for the security-relevant ones, anyway). A problem is that the permissions necessary for the program to run are easier to determine than the permissions necessary to protect the security of the system. Also, an artifact of the complexity of MAC policies is that users of such systems are no longer capable of modifying the OS distributors' policies. This may be a blessing in disguise, as this removes the responsibility of MAC policy specification from system administrators and places more demands on the OS distributors to assess programs. We have not yet seen the benefit from the former, as MAC policies for conventional systems are not designed to meet a security goal (e.g., Biba integrity [7], Clark-Wilson integrity [12], or even any practical approximation [28, 53, 57]) and OS distributors still lack the tools necessary to understand how a program's implementation may impact the security of the system at large.

Application Developers Can Follow Directions A variety of software engineering methodologies have been developed, but it was not until recently that security improvement became a focus. Meta-compilation [68], ITS4 [61], and Prefast [34] were developed to find program bugs, including security bugs. However, such tools are unsound (i.e., do not find all bugs). More powerful approaches were developed to prove the correctness of complex software [47], such as drivers, although such techniques do not scale to large software components. We believe that programmers could be induced to follow a testing approach that scaled to the size of systems effectively. Many companies implement structured test procedures before releasing code, but that has not had a tangible effect on overall system security. Studies have shown that "test-driven development" [39] does have a significant impact on defect reduction, but our concern is that we are not doing the right testing in programs nor are we testing the composition of programs and OS distributions into systems.

Emerging Systems Architectures Might Enable Better Scalability in Administration Say what one will about whether to trust your security-critical data to cloud systems [48], but the cloud architecture offers an opportunity to improve the scalability of system administrator decision-making. This occurs in two ways: (1) the cloud base platforms are defined by the cloud vendor, enabling a single configuration to apply to many systems, and (2) cloud vendors often provide a list of preconfigured OS distributions for their clients, aiming to encourage the use of known systems. In the first case, we envision that a single group of developers and administrators could define and manage the most secure system we can configure. Further, as this group could include the skills of application developers, OS distributors, and system administrators, they could cover the entire scope of security decisions. At present, however, few if any concrete security guarantees are offered by cloud vendors[8]. Second, by standardizing the OS distributions used, there is the potential that their configurations could be carefully designed to improve security. This will depend greatly on how well the OS distributors understand their programs' security defenses (or lack thereof).

Our claim is that the lack of a comprehensive methodology for designing and deploying systems that meet concrete security requirements prevents us from getting the upper-hand on attackers. As a result, everyone is the enemy of security. We have been looking for short-cuts, hoping that by incremental changes, we may be able to luck-out into the deployment of well-defended systems. However, the reality is otherwise. We need an approach based on concrete security goals. To date only information flow security models offer a precise and comprehensive understanding of possible attacks, as all the paths that attackers can use to access processes are identified by information flows. We need application developers to build their programs in such a way that they can understand the threats to their programs and evaluate the effectiveness of their defenses to such threats. We need OS distributors to be able to reason about program security in the context of their access control policy to determine if the threats they face are adequately addressed. Finally, we need a methodology where system administrators can make the decisions that they understand and leverage the improved efforts of the OS distributors and application developers effectively.

3 Attack Surfaces

We propose that the basis for security decisions should be the system's attack surface. An *attack surface* is defined as the entry points that are accessible to an attacker [20]. An attack surface was originally defined in the context of a program, but we use it in the context of programs, OS distributions, and systems, such that every design and deployment decision must account for the resulting attack surface and whether that attack surface can be adequately defended.

The key challenge regarding attack surfaces is to identify all the attack surfaces that may be used in a deployment. Consider the Apache web server program. The `httpd-2.2.14` distribution including the Apache core and all the modules contains 2451 unique library calls. In theory, any of these library calls may cause an Apache process to input data from a system object (e.g., a network connect, an IPC, a file, etc.) accessible to an attacker. However, we may not want just any interface to be used to read data that may be modified by an attacker. Instead, the Apache team may consider only the interfaces that are known to be accessible to attackers, Apache's attack surface. A problem is that the Apache team's view of an attack surface may not correspond to the actual attack surfaces created when it is deployed.

Probably, the Apache team will consider the network interfaces among its attack surface, but an Apache process may also retrieve untrusted inputs from files, IPCs, etc. For example, users may be able to provide content, including scripts, that Apache uses.

[8] As a contrast, concrete claims regarding physical security, such as armed guards for the data center, are made [3].

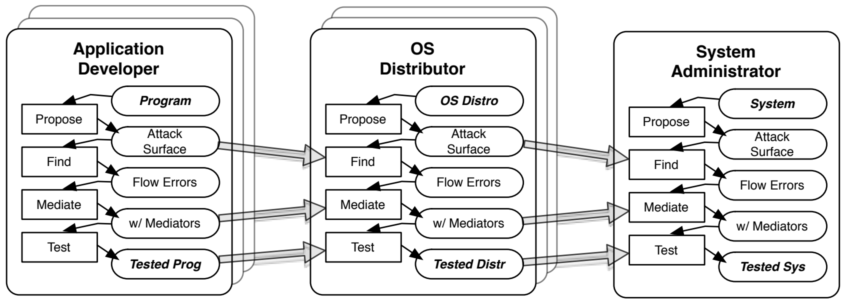

Fig. 2: The proposed approach for testing the composition of programs and OS distributions into systems.

In another case, when Apache forks a child process it creates a pipe to receive input from that child, but that child may be used to execute untrusted content, so such pipes may be the source of untrusted input. A recent vulnerability was found for this interface.

Previous work in identifying attack surfaces has focused on one component at a time, either on a program or an OS access control policy, but neither alone is sufficient to reason about attack surfaces accurately. First, researchers examined programs to identify possible attack surface interfaces and evaluate their significance [32]. In general, any program interface may define a location through which an attack may originate, so this work focused on identifying interfaces to valuable resources for attackers, hypothesizing that these would require the most attention for defense.

Second, others have used the system's access control policy to identify the programs that may have attack surfaces [11]. In this case, Linux systems with SELinux [43] and AppArmor [41] access control policies were compared based on the number of programs that were accessible to a network attacker and also had direct access to modify the Linux kernel (e.g., install a rootkit). What we want to know is whether attack surfaces created by the deployment conflict in some way with the expected attack surfaces of the program. For example, the OpenSSH daemon was carefully reengineered (privilege-separated) to limit the interfaces through which it receives untrusted input [45]. Nonetheless, a recent vulnerability was found caused by the incorrect parsing of users' authorized keys files. By looking at attack surfaces in the context of their deployment, we could locate this interface as a potential risk, rather than waiting for the attacker to identify it for us [62]. We also believe that examining OS distributions to identify attack surfaces in the context of their deployment, relative to network policies, is necessary to ensure that all decisions are acceptable for security.

4 Proposed Approach

What we want is to be able to test: (1) whether the attack surface expected for each component is consistent with its deployment and (2) that each component only permits authorized operations for its deployment. First, suppose that application developers constructed their program assuming that a program interface was adequately defended, but a vulnerability is found. In that case, OS distributors better design access control policies that prevent attackers from accessing that program interface. Second, the OS distributor must verify that the program when limited to this restricted attack surface only authorizes operations approved by the OS distributor.

The current approach to testing and the use of results of prior testing is inadequate to build secure systems. We surmise that program vulnerabilities are caused because: (1) the product was tested under an attack surface that differs (if any was identified) from the attack surface when deployed; (2) the product was not tested thoroughly enough to protect itself from threats at the "tested" interfaces; and/or (3) the OS distributors and system administrators do not know the extent of such testing, so their suggested deployment is a blind risk. A similar relationship between OS distributors and system administrators holds for determining whether an entire OS distribution will be deployed securely. Unfortunately, the testing of access control policies is even more ad hoc than for programs, as conventional systems that use MAC enforcement typically aim for least privilege [51], but that requirement cannot be precisely specified and tested. As a result, we are not surprised that the security community is in a reactive, rather than proactive mode of operation. The question is whether we can develop a methodology for *comprehensive testing of programs and distributions based on explicit assumptions that can be validated by parties that use these components.* We examine the roles that application developers, OS distributors, and system administrators would play (enabled by automated tools) to enable such a methodology.

Figure 2 shows the high-level view of the proposed approach. For each component, we see a sequence of steps consisting of: (1) (propose) propose the component's attack surface; (2) (find) identifying data (information) flows where an attacker affects the component's integrity, identifying a *flow error*; (3) (mediate) asserting *mediators* that comprehensively resolve all flow errors in the component; and (4) (test) testing the efficacy of the mediators to thwart all instances of the possible attacks from those flows. The aim is that each party inputs their component to this methodology and the methodology generates a security-tested version of that component. Any other party that uses a security-tested version obtains the proposed attack surface used in testing (from step 1), a summary of the information flows enabled by the component (from step 3), and the testing methodology used in determining the summary (from step 4). Thus, parties can test the use of others' components in their systems to obtain a comprehensive evaluation of security.

What we find intriguing is that the sequence of steps for each component is the same in this methodology, regardless of whether the component is a program, OS distribution, or system consisting of many OS distribution instances (hosts).

Application developers need to test their programs against the threats of attackers. First, they propose an attack surface for their program, which defines their assumption of the threats that are possible. An attack surface identifies low-integrity sources to the program. Second, application developers need to find what problems exist in their programs. In this case, a problem is where data from an attacker source may be used to modify data that is used for a high-integrity sink. That is, there is a data flow from the attacker to high-integrity program data. Third, the application developers must assert *mediation statements* to control such data flows. The problem is to determine what mediation statements to place at what locations in the code to address the illegal flow. The placement problem is non-trivial, as the application developer must make sure that all illegal flows are covered, this is difficult to ensure manually. Determining what mediation is necessary to prevent attacks is also difficult because such mediation is program-specific, in general. Fourth, we need to test that these mediations are adequate for the program. In general, testing is language and program-specific, but there is a rich literature in both testing methods and techniques for various problems, including security [5, 27, 34, 54, 68, 70]. We envision that our methodology will leverage such methods and techniques, making them available to application developers. Eclipse [16] is a good foundation for integrating such testing for programmers.

We find that OS distributors have to perform similar tasks as those of application developers, but they construct a different artifact (an OS distribution) and they build this from others' components[9] (multiple programs). As mentioned in Section 2.2, a key trend is that OS distributors will have some prior knowledge about the programs (security-relevant ones) that run on their systems, so they will configure MAC policies for the programs running on their systems. For each program, the OS distributor first proposes an attack surface for their distribution. This typically consists of identifying the networked programs on their system, but some systems enable inter-VM communication via hypervisor operations, resulting in more operations to consider. Second, the OS distributor needs to compute the programs that may be accessible to attackers, particularly how attackers may impact how the valuable system data may be accessed. In this case, an information flow analysis is proposed to find how network processes create information flows in the distribution (e.g., computed from SELinux policies [19, 22, 52, 59]). Rather than just using the access policy though, we envision that the information flows enabled by programs (from their data flows computed above) should be used to compute more accurate flows – not all program data flows, however, but a summary that expresses the information flows generated. Third, the OS distributor must identify where to resolve such information flow errors. In Biba integrity [7], such flows may be mediated by guards, but in conventional systems, the programs are expected to mediate their low-integrity inputs. In this case, the OS distributor needs information to make decisions about how to choose among such options based on what component attack surfaces they will accept. Fourth, the OS distributor must test the effectiveness of the selected mediation. Where such mediation depends on a program, we should leverage prior program testing in this evaluation.

Finally, the system administrators must configure systems consisting of one or more OS distributions into a coherently-defended whole. As described previously, system administrators' main focus is network policy (e.g., firewall). The question is how can configuration of a network policy build effectively on the work of the application developers and OS distributors. First, system administrators define the actual attack surface of the system from the network policy. However, they may not have as clear an understanding of what is valuable in the distributions that they use. Currently, OS deployments mix data (which belongs to the system administrator's organization) with code (which belongs to the OS distributor), so making this separation explicit and enabling each of the OS distributor and system administrator to define valuable data would make the security problem clearer. Second, while system administrators do not need to make any assumptions about what the attack surface might be, they must determine

[9] Of course, application developers may have to compose programs from others' programs, so in those cases, they may have to adopt some aspects of the OS distributors' tasks described here.

how the network policy enables attacks across hosts. A summary of the information flows generated by OS distributions, including their programs, should be constructed from which system-level information flows can be computed to identify information flow errors. Third, system administrators may need to assert network mediation, such as the placement of firewall rules to control information flow among individual distribution instances. At present, this seems like something that system administrators can do relatively well, but we may find that more accurate configuration of programs and systems may expose limitations in manual network configuration. Fourth, the system administrators must test the resultant configuration. A key ingredient in such testing is that it builds on the previous testing of OS distributors and application developers, but also informs system administrators of discrepancies between assumptions in attack surfaces that underlie any error.

5 Deploying the Approach

In this section, we propose that the *hierarchical state machine* (HSM) model [2] can serve as the formal foundation for our approach. First, we show that data flow in programs, distributions consisting of multiple programs, and systems consisting of multiple distributions can be expressed using an HSM. As a result, we can annotate an HSM representation with assumptions about attack surfaces, basically untrusted inputs to the representation. We then discuss the problem of inferring and resolving flow errors using data flow analyses of an HSM instance, based on the approach of the previous section.

5.1 Hierarchical State Machine Model

First, we define the hierarchical state machine (HSM) model [2].

Definition 1 *A* hierarchical state machine K *is a tuple* $(K_1, ...K_n)$ *of* modules*, where each module* K_i *has the following components:*

- *A finite set* V_i *of* vertices*, and a finite set* B_i *of* boxes.
- *Subsets* I_i *and* O_i *of* V_i*, respectively consisting of the* entry vertices *and* exit vertices.
- *An* indexing function $Y_i : B_i \rightarrow \{i+1, ..., n\}$ *that maps each box of the i-th module to an index greater than* i*. That is, if* $Y_i(b) = j$ *for box* b *of module* K_i*, then* b *can be viewed as a reference to the definition of module* K_j.
- *If* b *is a box of the module* K_i *with* $j = Y_i(b)$*, then pairs of the form* (b, u) *with* $u \in I_j$ *are the* calls *of* K_i *and pairs of the form* (b, v) *with* $v \in O_j$ *are the* returns *of* K_j.
- *An* edge relation E_i *consisting of pairs* (u, v)*, where the source* u *is either a vertex or a return of* K_i *and* v *is either a vertex or a call of* K_i.

An HSM model represents an hierarchical structure of *modules* connected by interfaces, called *boxes*. Modules may have an arbitrary internal structure of connections, so they are represented by a graph. The HSM model is a well-known formalism in the model checking community, and we leverage it because it gives us a well-understood formalism on which to base our analysis.

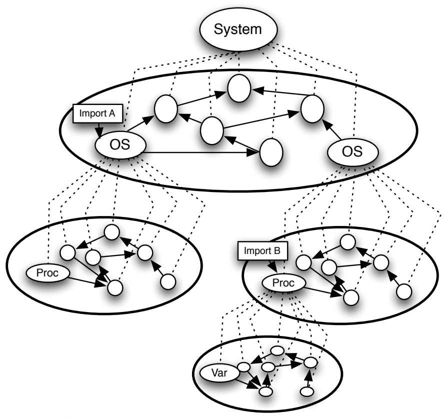

Fig. 3: **A system of hosts (OS) and programs (proc) create an encapsulated, hierarchical system of information flows.**

We find that the HSM model maps directly to that of a system of hierarchically-arranged components, as ours is. We describe the intuition here, shown in Figure 3. What is important about the structure of such systems is that they are encapsulated, hierarchical systems. A system is *encapsulated* in that all interaction between components must be mediated by their reference validation mechanisms. Program information flows can only be propagated to other programs through operating system mechanisms. The hosts (i.e., instances of distributions) can only communicate via the network or virtual machine mechanisms (if on a VM system). These systems are also *hierarchical* in that the authority to make security decisions is monotonically-reduced from the root to the leaves. For example, processes cannot make a security decision unless their operating system authorizes them to make that decision.

Converting a program, distribution, or system to an HSM representation involves identifying each component that enforces its own information flow security policy, computing the authorized information flows of that component, and connecting the information flows between parent and child components. Figure 3 shows the resultant representation for a system. At the leaves are the programs that enforce information flow security. These programs include all the programs that have any attack surface. These programs must, at a minimum, ensure that low integrity data that they receive is sanitized effectively, although mandatory access control within programs is also practical now [38, 58]. Next, each OS distribution enforces its own access control policy, so if such a policy represents an enforcement of information flow security then it can be converted to an HSM module. Such policies must be mandatory access control policies that can be converted to an information flow representation, such as information flow policies [7, 18], type enforcement policies [43, 63], approximations of information flow enforcement [28, 53, 57], or sandbox policies that confine enough processes to constrain information flow [41]. Dynamic information flow enforcement, such as the Decentralized Information Flow Control model [25, 69], may be converted to an HSM model, although program discretion about the creation of new attack surfaces must be modeled. Finally, virtual machine monitor and/or network policies control information flow among individual hosts in a system. Figure 3 shows just one level of hosts, where

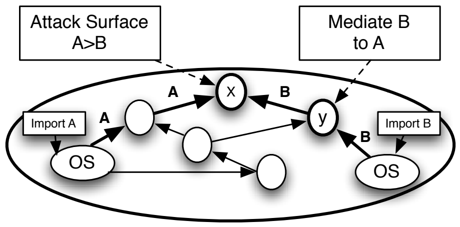

Fig. 4: **A simple information flow graph where valuable data of integrity level A and attack data of integrity level B are imported. Vertex x has an attack surface caused by the graph's flows that is mediated at vertex y.**

system policy controls information flow among hosts, using firewalls or VMM policies. For example, firewalls state communication paths between hosts, and VMM policies state which virtual resources may be shared by hosts running in VMs [13, 26, 50].

The most difficult task is to connect the information flows between parent and child modules. In constructing an HSM instance from a set of policies, we need to know how components at a parent layer (e.g., system) are connected to components at a child layer (e.g., hosts) and construct boxes to represent such connections (see Definition 1). This is fairly well-defined between the system and hosts by firewall and VMM policies. For example, firewalls state which ports can receive a packet, but many ports have well-known associations with processes. Nonetheless, such information may be ambiguous, so the use of labeled networking policies [21,35,42] is recommended, as these explicitly state the security labels of the processes that are authorized to use network connections. For VMMs, often any root process is typically authorized to make hypervisor calls, so these processes must be connected to the inter-VM flows authorized by a VMM policy. Fortunately, the only normal inter-VM interaction is between guest VMs and a single privileged VM, although this makes the privileged VM more complex. Finally, for programs, any program interface is authorized to access any resource that its process is authorized for (based on the process's security label). This presents a problem in that any interface may be part of that program's attack surface, although this is typically not expected to be the case. The HSM model makes this relationship explicit, and our approach aims to tease out the actual program attack surfaces.

5.2 Proposing Attack Surfaces

Assuming an attack surface for a program, distribution, or system involves identifying where an attacker may access that component. While the notion of "assuming" an attack surface is inherently incomplete and subject to change, it is important to state the assumptions under which one makes security decisions for a component. Others can then use a component under those assumptions or cause the assumptions to be re-evaluated. Neither task is performed in any principled manner currently.

An attack surface has a similar meaning for each component, but a different physical manifestation. All attack surfaces refer to the sources of untrusted data to the component. In the Common Criteria, these are called *imports*[10]. For a program, its attack

[10] Exports are also a concern for secrecy.

surface consists of program interfaces (code instructions) that import data that may be modified by an attacker. For a host, its attack surface consists of a set of processes that have access to attacker data (e.g., data imported via devices, such as the network or disk, or program data downloaded with the distribution). For a system, its attack surface consists of the untrusted hosts and external components accessible to the system (e.g., via the network). Finally, we note that components also have valuable data, so imports must also identify key valuables (otherwise the attacker has nothing to attack). Elliciting imports with minimal manual effort is the goal, and we are exploring the development of a knowledge base to infer a conservative set of imports and their relative integrity relationships automatically [62].

The simple view of an attack surface in an HSM instance is shown in Figure 4. This figure shows that all these surfaces are represented by a set of imports to the graph, which represent set of entry vertices from boxes of its potential parent components. Imports are explicitly added to the HSM model to show where attacker data (level B) and valuable data (level A) originates.

5.3 Finding Flow Errors

Once we know we know where attacker data and valuable data are imported into the component, we need to identify flow errors, cases where an attacker can impact the component in unauthorized ways. While the security community has significant experience in inferring information flows and detecting information errors in programs and systems, we find that inferring information flow errors from OS attack surfaces may require different methods than for programs.

An information flow error occurs where a component tries to access unauthorized (i.e., lower integrity) data. For programs, information flows are inferred based on Denning's lattice model [15]. In this model, a component may either be bound to a security class (e.g., integrity level) statically or dynamically. If a component is bound to a security class statically and an access it performs violates the authorized flows in the lattice, then that access is a information flow error. If the component is dynamically bound to a security class, then the security class of the accessed data is combined with the current security class (e.g., using a least upper bound for the lattice) to assign a new security class to the component, if necessary. An error may then occur when the dynamically bound component is used by a component with a statically bound security class. This method of inference has been applied to identify information flow errors for both secrecy and integrity in programs [37].

For OS information flow policies, the bindings are typically all static (Biba and Clark-Wilson integrity [7,12]) or all dynamic (LOMAC [7,18]), although the IX system allows dynamic binding with limits [33]. Thus, typical OS integrity policies bind or set bounds on each process or object *a priori*, or the entire OS is dynamically bound with no constraints. In conventional systems, integrity levels are not bound to each process or objects, so it is necessary to determine what each is supposed to be. However, manually binding each process or object to an integrity level is impractical, so the nature approach would be to bind some statically (e.g., attackers, which are low, and trusted components, which are high), and dynamically bind the rest using Denning's inference approach. Then, information flow errors can be found.

However, we find that this inference approach does not work well for non-information flow, OS policies, such as Type Enforcement [8]. If we bind processes that access the network to low integrity and the kernel to high integrity, we find that almost all the dynamically bound processes will be inferred to be low. This may be an accurate representation of modern, conventional operating systems, but we want to identify problems and fix them coherently. As an alternative, we propose an inference model where each process's integrity is determined by the transitive closure of the integrity levels that reach it [49]. This approach reflects a process's desire to remain high integrity, unless explicitly downgraded, but still shows that all processes are insecure (i.e., receive data of multiple integrity levels), unless mediation is performed somewhere. Using this inference approach, flow errors correspond to processes or objects (labels in a MAC policy) in the OS access control policy that receive multiple integrity levels of data.

Figure 4 shows how the HSM model is used to compute information flow errors. A layer-specific method is used to propagate imported integrity levels through the system. In practice, the application developer will focus only on their programs and the flow errors identified at that level, using Denning's inference. A number of programming languages have tools that provide such inference, including Java [38], OCaml [55], and C [11]. OS distribution and system layer analysis is either performed based on an information flow policy or based on the transitive closure approach above for a non-information flow policy. In Figure 4, transitive closure propagation shows that A and B both reach vertex X.

5.4 Mediating Flow Errors

Once flow errors are found, then the respective parties need to resolve such errors. The first decision is which party is assigned to fix the problem. For example, if the problem is a mismatch between the deployment attack surface and that which was assumed by the child component, then the question is whether to fix the policy or the attack surface. We assume that this decision has been made. Traditionally, resolution is a manual process, but the aim is to leverage the HSM model to automate some steps of the resolution process.

We have developed a method that generates placements for resolving data flow errors based on graph cuts [24]. In this method, the program is converted into an information flow graph, as above, and all paths from a source to a program location where a flow error occurs must be cut by a mediation statement (e.g., sanitizer or runtime check) that resolves that error. This method is general for information flow graphs, so we propose to apply it to information flow graphs at different layers in the HSM model like the OS distribution layer. Figure 4 shows how a flow of level B is cut at vertex y removing all attack surfaces (not just the one shown at vertex X). However, for the approach to be practical, we must be able to construct complete cut problems. For this, we need to know the mediation statements, their possible locations, and costs (we can then use a min-cut algorithm). We also need to account for functionality, as we cannot simply cut access control policy flows, as they may be necessary for the system to work.

[11] Since C is not type-safe, such inference only applies if the application developer is not adversarial.

An advantage of reasoning about security in terms of information flows is that this reduces the number of options for resolving a flow error. We can either change a vertex or change the flows into or out of a vertex. Changes to vertex include removing it (e.g., removing a program from a distribution) or changing the information flows within the vertex (e.g., changing attack surface of the program). Changes to an edge include adding a sanitizer on an edge (e.g., *guard* process) or changing the integrity level of the imports (e.g., use more reliable inputs). We note that removing an edge alone is problematic in that this is likely to remove a necessary function for that vertex.

The problem of automating resolution is complex due to conflicting constraints. A system may both *require* particular information flows to occur for the system to function and *restrict* certain information flows from occurring. Finding solutions to constraint systems with positive and negative constraints, in our setting, will result in a PSPACE-hard branching-time model checking problem [2]. As a result, even with a small number of resolution options, if there are a large number of possible locations, then automating resolution will be computationally complex. Powerful solvers (e.g., SAT solvers) are now available that can search large solution spaces efficiently.

If the solution is to add a sanitizer or runtime check at a particular location, then the question is what this code should do. Historically, sanitization has been error-prone, so identifying locations is not sufficient to ensure error resolution. Further, researchers have found that simply placing one sanitizer may not be sufficient as different uses may require different sanitizations [5]. Thus, the purpose of the resolution must be clear enough to assess whether other mediation may also be required to satisfy security constraints. For example, if a sanitizer is for a web server to handle untrusted data, then that mediation may still permit data that is unsafe for the database (e.g., for SQL injections). Subsequent resolutions must be found for these "secondary" imports resulting from partial sanitization. Finally, while sanitization is inherently program-specific, a number of sanitizing functions have been identified over based on the type of error, programming language, etc. Tools based on this methodology should provide access to known sanitization functions to reduce the effort of the parties.

5.5 Testing the Resulting System

Finally, once the actor has decided on a resolution to any flow errors, it is necessary to test such resolutions, particularly for sanitization. Sanitization aims to allow a transformed version of an information flow that meets some requirements. Such requirements must be made explicit and test thoroughly. Fortunately, a variety of methods for testing sanitization procedures have been proposed, although most are language and bug-specific [5,64]. We envision that such procedures would be provided to application developers and sanitizer developers for systems, and that a required degree of testing would be enforced.

Testing of sanitizers involves conservative static analysis to ensure no false negatives (errors relative to requirements) and a supplementary runtime analysis to validate the existence of real errors [5]. Such testing is limited by the problem of identifying the sources of untrusted data, but the HSM model makes the sources of untrusted data explicit. Also, test cases need to be generated for the runtime validation. Fuzz testing tools generate inputs to programs to find vulnerabilities. One tool, EXE, automatically generates inputs that will crash a program [9]. It tracks possible values that variables can have symbolically. When an `assert` statement is reached, EXE checks if there is an input that causes the statement to become false.

6 Conclusions

Developing a software engineering methodology for security would be a significant undertaking. In addition to providing mechanisms to convert the relevant program, distribution, and system information into a canonical format (an HSM instance), algorithms must be developed to solve the problems highlighted above, and a user interface must be designed to convey this information clearly. Finally, testing tools must be integrated with the methodology to enable comprehensive testing for all the target languages and bugs.

Waiting for a "big bang" of all the technology above before we have a useful system pretty much guarantees that it will never happen, so the question is how should we proceed to provide a useful, but perhaps incomplete, functionality that leads to a desired goal. We envision that such tools must be integrated into a common, open software engineering ecosystem, such as Eclipse. We imagine that any initial methodology would enable testing of one component and the testing of its deployment, such as building a program and testing its deployment in an OS distribution. Finally, the security community will have to consider how to pull together the myriad of prior research into a coherent approach, whether for the proposed approach or another. The security community has undertaken similar challenges for defining assurance criteria, and this will be a similarly large undertaking.

References

1. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of CCS '05*. ACM, 2005.
2. R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3), 2001.
3. Amazon. Amazon Web Services Security Center. `http://aws.amazon.com`.
4. J. Ames, S. R., M. Gasser, and R. R. Schell. Security kernel design and implementation: An introduction. *Computer*, 16(7):14–22, 1983.
5. D. Balzarotti *et al.* Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.
6. The Bastille hardening program: Increased security for your OS. `http://bastille-linux.sourceforge.net`.
7. K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE, April 1977.
8. W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th NCSC*, 1985.
9. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.
10. M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of OSDI '06*. USENIX Association, 2006.

11. H. Chen, N. Li, and Z. Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of NDSS '09*, 2009.
12. D. D. Clark and D. Wilson. A Comparison of Military and Commercial Security Policies. In *1987 IEEE Symposium on Security and Privacy*, May 1987.
13. G. Coker. Xen Security Modules (XSM). `http://www.xen.org/files/xensummit_4/xsm-summit-041707_Coker.pdf`.
14. C. Cowan *et al.* Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symp.*, 1998.
15. D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5), 1976.
16. Eclipse. `http://www.eclipse.org`.
17. H. Feng *et al.* Formalizing sensitivity in static analysis for intrusion detection. *Proceeding of the 2004 IEEE Symposium on Security and Privacy*, 0, 2004.
18. T. Fraser. LOMAC: MAC you can live with. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, June 2001.
19. J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying Information Flow Goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1), 2005.
20. M. Howard, J. Pincus, and J. M. Wing. Measuring Relative Attack Surfaces. In *Proceedings of Workshop on Advanced Developments in Software and Systems Security*, 2003.
21. T. Jaeger, K. Butler, D. H. King, S. Hallyn, J. Latten, and X. Zhang. Leveraging IPsec for Mandatory Access Control Across Systems. In *Proceedings of SecureComm '06*, Aug. 2006.
22. T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symp.*, Aug. 2003.
23. P. Karger, M. Zurko, D. Bonin, A. Mason, and C. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Trans. Softw. Eng.*, 17(11), 1991.
24. D. King *et al.* Automating security mediation placement. In *Proceedings of ESOP '10*, pages 327–344, 2010.
25. M. N. Krohn *et al.* Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM SOSP*, Oct. 2007.
26. KVM: Kernel based virtual machine. `http://www.linux-kvm.org`.
27. D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
28. N. Li, Z. Mao, and H. Chen. Usable Mandatory Integrity Protection For Operating Systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, May 2007.
29. C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th conference on USENIX Security Symposium*, 2005.
30. S. B. Lipner. Non-discretionery controls for commercial applications. *Proceedings of IEEE Symposium on Security and Privacy*, 0, 1982.
31. P. Loscocco *et al.* The Inevitability of Failure: The Flawed Assumptions of Security Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, 1998.
32. P. Manadhata, K. Tan, R. Maxion, and J. M. Wing. An Approach to Measuring A System's Attack Surface. Technical Report CMU-CS-07-146, School of Computer Science, Carnegie Mellon University, 2007.
33. D. McIlroy and J. Reeds. Multilevel windows on a single-level terminal. In *Proceedings of the (First) USENIX Security Workshop*, Aug. 1988.
34. Microsoft. Prefast for drivers. `http://www.microsoft.com/whdc/devtools/tools/prefast.mspx`.
35. J. Morris. New secmark-based network controls for selinux. `http://james-morris.livejournal.com/11010.html`.

36. MSDN. Mandatory Integrity Control (Windows). `http://msdn.microsoft.com/en-us/library/bb648648%28VS.85%29.aspx`.
37. A. C. Myers and B. Liskov. A decentralized model for information flow control. *ACM Operating Systems Review*, 31(5), Oct. 1997.
38. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. `http://www.cs.cornell.edu/jif,July2001-2003`.
39. N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Softw. Engg.*, 13(3):289–302, 2008.
40. J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of NDSS '05*, 2005.
41. Novell. AppArmor Linux Application Security. `http://www.novell.com/linux/security/apparmor/`.
42. NetLabel - Explicit labeled networking for Linux. `http://www.nsa.gov/selinux`.
43. Security-Enhanced Linux. `http://www.nsa.gov/selinux`.
44. PaX homepage. `http://pax.grsecurity.net`.
45. N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symp.* USENIX Association, 2003.
46. F. Qin *et al.* Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of MICRO '06*, 2006.
47. M. Research. SLAM - Microsoft Research.
48. T. Ristenpart *et al.* Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM CCS*, 2009.
49. S. Rueda, H. Vijayakumar, and T. Jaeger. Analysis of virtual machine system policies. In *Proceedings of SACMAT '09*, 2009.
50. R. Sailer *et al.* Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *Proceedings of ACSAC '05*, 2005.
51. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), September 1975.
52. B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 WITS*, April 2004.
53. U. Shankar, T. Jaeger, and R. Sailer. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Proceedings of the 2006 NDSS*, February 2006.
54. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symp.*, 2001.
55. V. Simonet. The Flow Caml System: Documentation and User's Manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003. ©INRIA.
56. J. Solworth. Robustly secure computer systems: A new security paradigm of system discontinuity. In *Proceedings of NSPW '07*, 2007.
57. W. Sun *et al.* Practical proactive integrity preservation: A basis for malware defense. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.
58. Tresys. Selinux userspace. `http://userspace.selinuxproject.org/trac/`.
59. Tresys. SETools - Policy Analysis Tools for SELinux. Available at http://oss.tresys.com/projects/setools. `http://oss.tresys.com/projects/setools`.
60. S. Vidyaraman, M. Chandrasekaran, and S. Upadhyaya. The user is the enemy. In *Proceedings of NSPW '07*, 2007.
61. J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. Token-based scanning of source code for security problems. *ACM Trans. Inf. Syst. Secur.*, 5(3):238–261, 2002.

62. H. Vijayakumar *et al.* Integrity walls: Finding attack surfaces from mandatory access control policies. Technical Report Technical Report NAS-TR-0124-2010, Network and Security Research Center, Feb. 2010.
63. K. M. Walker *et al.* Confining root programs with domain and type enforcement (DTE). In *Proceedings of the 6th USENIX Security Symp.*, 1996.
64. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. *SIGPLAN Not.*, 42(6):32–41, 2007.
65. Wietse Venema. Postfix Architecture Overview. `http://www.postfix.org/overview.html`.
66. A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004.
67. G. Wurster and P.C. van Oorschot. The developer is the enemy. In *Proceedings of NSPW '08*, 2008.
68. J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.
69. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the* 7^{th} *OSDI*, 2006.
70. X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symp.*, 2002.