On Risk in Access Control Enforcement

Giuseppe Petracca
The Pennsylvania State University
School of Electrical Engineering and Computer Science
gxp18@cse.psu.edu

Frank Capobianco
The Pennsylvania State University
School of Electrical Engineering and Computer Science
fnc110@cse.psu.edu

Christian Skalka
The University of Vermont
College of Engineering and Mathematical Sciences
skalka@cs.uvm.edu

Trent Jaeger
The Pennsylvania State University
School of Electrical Engineering and Computer Science
tjaeger@cse.psu.edu

ABSTRACT

While we have long had principles describing how access control enforcement should be implemented, such as the *reference monitor concept*, imprecision in access control mechanisms and access control policies leads to risks that may enable exploitation. In practice, least privilege access control policies often allow information flows that may enable exploits. In addition, the implementation of access control mechanisms often tries to balance security with ease of use implicitly (e.g., with respect to determining where to place authorization hooks) and approaches to tighten access control, such as accounting for program context, are ad hoc. In this paper, we define four types of risks in access control enforcement and explore possible approaches and challenges in tracking those types of risks. In principle, we advocate runtime tracking to produce risk estimates for each of these types of risk. To better understand the potential of risk estimation for authorization, we propose risk estimate functions for each of the four types of risk, finding that benign program deployments accumulate risks in each of the four areas for ten Android programs examined. As a result, we find that tracking of relative risk may be useful for guiding changes to security choices, such as authorized unsafe operations or placement of authorization checks, when risk differs from that expected.

CCS CONCEPTS

•**Security and privacy** → *Operating systems security;* Access control; Software security engineering;

KEYWORDS

Risk, Access Control Enforcement

ACM Reference format:
Giuseppe Petracca, Frank Capobianco, Christian Skalka, and Trent Jaeger. 2017. On Risk in Access Control Enforcement. In *Proceedings of SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA*, , 12 pages.
DOI: http://dx.doi.org/10.1145/3078861.3078872

1 INTRODUCTION

Access control restricts the subjects (e.g., users and programs) that may perform operations (e.g., read and write) over objects (e.g., files and records). It has long been recommended that the software that implements access control should separate the access control mechanism from the access control policy. Access control mechanisms should satisfy the requirements specified by the reference monitor concept [6], such as complete mediation, to enforce access control policies correctly. Access control policies should express the access control requirements to be enforced.

In practice, the design and implementation of both access control mechanisms and policies is a manual process, which leads to risks that adversaries may circumvent the access control goals. First, access control policies may allow unsafe operations. While multilevel security (MLS) policies, such as Bell-La Padula [7] and Biba [8], block risky information flows, MLS policies often result in many trusted subjects (i.e., trusted readers and writers) that are outside the policy [27]. In addition, least privilege policies [43], which favor functionality over blocking risky operations, are more popular today. Also, authorized subjects may be compromised by adversaries, introducing risk even for accesses that would have been safe for the uncompromised subject. Second, errors in implementing access control mechanisms may produce risks. For example, programmers may misplace the access control policy checks, which are often called *authorization hooks*, in their programs. To reduce the number of authorization hooks needed, programmers may allow subjects to access more objects or perform more operations, possibly including risky operations with authorized operations. For example, one hook may allow two objects to be read, one of which may be used to create an information flow from other subjects. At present, there are no methods to track risks in overly coarse authorization hook placements.

Current research does not track the risk created by manual access control system design and implementation. Risk has generally been explored in two ways: (1) to identify undesirable permissions in access control policies statically and (2) to produce access control models that integrate risk estimates into the authorization decisions. First, researchers have proposed static analysis tools for mandatory access control policies to identify permission assignments that may impact the secrecy and integrity of the system [3, 10, 18, 28]. For example, Jaeger *et al.* identify permission assignments that violate Biba integrity as unsafe [28]. The point of such work to motivate changes in the policy design, but sometimes unsafe permissions are deemed necessary for functional reasons.

More recent work focuses on how applications may misuse the permissions assigned to them [20, 21, 30], such as when a mobile phone app leaks data [19]. Second, researchers have proposed access control models that integrate risk into the access control decision process [9, 11, 13, 31, 37, 42]. Bijon *et al.* [9] examine which relationships in the RBAC model may be made risk-aware and examine constraint-based and metric-based expressions of risk. Chen and Crampton [11] propose to leverage an estimate of the risk that a subject is not trustworthy or a permission assignment is not appropriate in authorizing an operation. However, estimating risk values or defining risks constraints on such concepts involves a significant amount of subjectivity regarding trustworthiness and appropriateness.

In this paper, we explore the problem of tracking the accumulation of risk in access control enforcement. Toward this goal, we are motivated by three types of efforts. First, researchers have demonstrated methods to compute trusted and untrusted resources for each subject [49]. By identifying trusted resources for each subject, we can identify operations that risk integrity for each subject. We will have to develop analogous principles for identifying resources that are secret to a subject. Second, researchers have demonstrated the value of auditing to enforce retrospective policies [50]. Access control cannot produce false positives, which implies that some risky operations may be allowed, but auditing may be leveraged to enforce such operations retrospectively to reduce risk. We explore auditing the risk of authorized, unsafe operations. Third, researchers have shown that risk may accumulate systematically for unsafe operations, as for differential privacy [16]. For each query to a differentially private database, an estimate on the privacy loss incurred by that query operation increments a risk that an adversary may infer the presence of a particular record in the database. In this work, we explore ways to estimate accumulated risk in the context of access control decisions. However, relating accumulated risk to security properties remains future work, as we explore different types of risks that may accumulate due to access control.

Based on these insights, we identify four different types of risks taken in access control: (1) risk due to authorizing unsafe operations; (2) risk due to abuse of authorized permissions; (3) risk due to lack of program context; and (4) risk due to the granularity of authorization hooks. We explore how to identify such risks, ways we may associate risk estimate values with such risks, and issues in computing such risk estimates in practice. We would then envision that access control mechanisms would record risk estimates as they accumulate using auditing both within the programs (to collect inputs for computing risk) and within the access control mechanism (to collect risks regarding enforcement) to enable risk computation. Ideally, accumulated risk may either lead to a denial or at least some change in access control enforcement, such as changes in policies or authorization hook placements, along the lines of preventing violations in differential privacy. Formalizing properties related to risk in access control enforcement in a manner analogous to differential privacy remains future work.

We evaluate our proposed approach to risk estimation in Android systems. First, we use the Android Compatibility Test Suite (CTS) to study risks due to unsafe operations and permission abuse in an Android 6.0.1 system protected by its SEAndroid policy. We find that high integrity Android system processes take only a modest number of risky operations to access data modified by untrusted apps, but each high integrity process accesses some risky objects. Also, we found that no program uses more than 12% of its permissions in CTS operation, indicating that detecting permission abuse may be practical. Second, we computed the risk estimates for ten Android system processes on the same Android system based on events generated by the Android UI/Application Exerciser Monkey. We found that all these processes accumulate a modest amount of risk along each risk dimension (less that 0.1 for our equations), indicating that risk estimates may be useful for comparing relative risks.

In summary, this paper makes the following contributions:

- We identify four risk areas in access control enforcement related to weaknesses in access control policies and weaknesses in how access control is enforced.
- We outline an approach for computing risk estimates and logging such estimates as security-sensitive operations occur that aims for monotonicity of risk over time for each subject and proportionality with respect to impact.
- We evaluate the four risk areas on programs in Android systems, finding that programs generally run with a small, but tangible, risk, which may be used to compare relative risks of programs.

The remainder of the paper is structured as follows. Section 2 describes the four types of risk we identify in access control enforcement. Section 3 specifies the security model we assume when assessing risk. Section 4 outlines our objectives for tracking risk. Section 5 examines runtime risk estimation, proposing techniques for risk estimation and identify challenges in gathering information necessary for risk estimation. Section 6 examines risk estimation in Android systems. Section 7 discusses some issues found in risk estimation. Section 8 concludes the paper.

2 RISKS IN ACCESS CONTROL

The fundamental problem is that choices in both access control enforcement mechanisms and access control policies introduce risk, and that at present we neither track nor react to such risk. As a result, adversaries may be allowed to perform operations that exploit such risks with impunity. Thus, if an adversary can find an exploit for the risks taken in a system deployment, then they may be able to abuse such an exploit persistently, as a so-called *Advanced Persistent Threat* (APT), possibly across many hosts.

We examine four classes of risks in this paper: (1) risk due to the authorization of unsafe operations; (2) risk due to the possible compromise of subjects; (3) risk due to enforcement context; and (4) risk due to the granularity of mediation. While these classes of risk are not entirely independent (e.g., allowing unsafe operations may be a result of the reference monitor design), we examine each class separately in this paper.

2.1 Access Control Enforcement

Before delving into the risks, we review the relevant principles of access control enforcement. Traditionally, access control restricts which subjects (e.g., users and processes) can perform which operations (e.g., read and write) on which objects (e.g., files and database

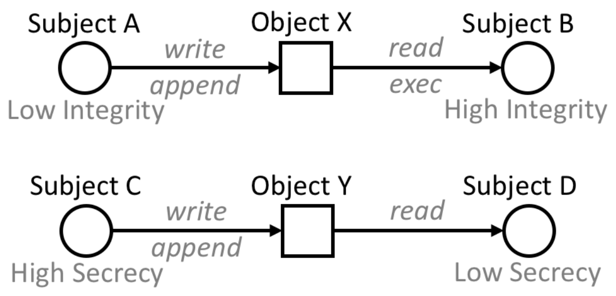

Figure 1: Illegal information flows for integrity and secrecy

records). Programs (e.g., operation system or database) are said to enforce access control using a *reference validation mechanism*. A reference validation mechanism consists of three components: (1) *authorization hooks*, which mediate security-sensitive operations in the program and produce authorization queries identifying the subject, object, and operation to be authorized; (2) *authorization mechanism*, which compares authorization queries to the access control policy to determine the authorization result; and (3) *access control policies*, which define the granted authorization queries.

The *reference monitor concept* [6] proposes three requirements for correct reference validation mechanisms. First, a reference validation mechanism must be non-bypassable. That is, the authorization hooks must mediate access to all security-sensitive operations, which is also called *complete mediation*. Second, the authorization mechanism must be verifiable. That is, it must be possible to test that the reference validation mechanism enforces the expected security policy. Third, the reference validation mechanism and access control policy must be tamperproof to prevent attacks on access control itself. Finally, although not explicitly stated by the reference monitor concept, the access control policy must correctly define the expected security requirements.

One challenge is to identify security-sensitive operations in programs. Researchers have proposed identifying security-sensitive operations via program data flows [15, 36] and control flows [22, 23, 34, 35, 44]. Using data flows, security-sensitive operations occur when a program data flow violates the program's information flow policy. For example, an assignment of a variable with secret data to a variable sent to the public network would be a security-sensitive operation because that data flow would violate an information flow requirement that secret data not be made available to public subjects. Using control flows, researchers have proposed using both syntactic and semantic features of programs to identify security-sensitive operations. One semantic approach [34] identifies security-sensitive operations by data-flow and control-flow "choices" that programs make with untrusted input. For example, if a program uses untrusted input in a conditional, the program makes a control-flow choice based on such untrusted input that should be mediated. Neither approach is perfect, as we would like to proactively mediate operations before information-flow violations (for data flow), but heuristics are currently necessary to identify security-sensitive operations before such violations occur (for control flow).

2.2 Risks of Authorizing Unsafe Operations

While an aim of computer security has long been to prevent the execution of unsafe operations by restricting *information flows* [15] for secrecy [7] and integrity [8] as shown in Figure 1. However, in practice, systems deployments take risks with respect to information flows. When information flow policies are employed, risks are taken in the ad hoc design of declassifiers and endorsers [36] or choice of trusted readers and writers. Most commercial systems employ *least privilege* [43], where the permissions available to each subject are determined by the functionality required of the subject. Researchers have shown that even mandatory access control policies based on least privilege produce integrity risks [28], even for highly privileged processes [10]. We will refer to a permission that violates an information flow constraint as a *unsafe permission*.

Unfortunately, static analysis of access control policies to highlight possible risks has not had a tangible impact on the design of access control policies. We have found that the stock Android policy for the Android 6.0.1 (kernel 3.4.0) version system allows all the high integrity subjects (i.e., those started directly or indirectly by Android kernel subjects) to read, and in some cases even execute, objects that can be modified by low integrity subjects, according to the default SE Android policy. For 57 high integrity subjects identified, they can read or execute 57 object labels that may be written by low integrity subjects in this policy and have over 22,000 permissions that are not available to low integrity subjects. Thus, to prevent unauthorized access by low integrity subjects, we must ensure that high integrity subjects are not compromised when utilizing an object assigned one of these 57 labels controlled by low integrity subjects. However, no systematic approach is taken either to protect the high integrity subjects that use such objects or track the risk created by their use.

In addition, vendors extend the stock Android policy for their own devices. In one vendor's system[1], the number of low integrity object labels accessible to high integrity subjects increases to 240 from 57, and over 100,000 permissions are available to high integrity subjects only. Thus, simply shaming policy designers regarding violations of information flow appears to be an insufficient approach.

More recently, researchers have developed risk models based on a variety of properties of applications, such as application ratings [12, 14] and descriptions [38], in addition to static permission assignments [18, 30, 39]. Such techniques tend to focus on differences between individual applications and average applications, sometimes limited to a specific class of applications. While some malicious applications may be identified as outliers, applications' normal least privilege policies often include permissions that enable a variety of attacks, as described above. Thus, closer tracking of applications' use of their permissions is necessary to detect malice. However, fine-grained tracking, such as dynamic taint tracking [19], incurs an overhead that prevents wide deployment.

2.3 Risks of Permission Abuse

Another problem is that processes may be compromised, and adversaries may utilize the compromised process's permissions freely. In addition, insiders may misuse their authorized permissions to leak information or embed malice in a system. Researchers identified

[1]We anonymize the vendor system.

this problem in defining role-based access control (RBAC) models that reason about risk [9, 11]. For example, Chen and Crampton integrate risk thresholds into access control decisions, enabling administrators to utilize knowledge of the trustworthiness of users, competence of users in roles, and the appropriateness of a permission for a role. However, a challenge is to quantify risk for these cases objectively.

An interesting analogue for this problem is the problem of differential privacy. Differential privacy is a method for limiting the risk associated with allowing authorized subjects access to sensitive data [16]. In this case, the sensitive data is a database of individuals' anonymized records, and the intuitive goal is to make the database indistinguishable from another database missing any one individual's records. Functionally, systems that enforce differential privacy [25, 33, 41] limit the queries that may be executed to a total privacy budget. Interestingly, researchers have proven the "cost" of a set of queries in terms of information leaked is bounded by the sum of the costs for each query.

While preventing information leakage is a common goal for both differentially private databases and access control enforcement, our ability to measure when the requests to an access control system have exceed a bound has not yet been formalized. However, methods have been proposed to estimate data leakage. McCammant and Ernst [32] define a method for estimating the amount of information leakage in a program statically, although one must identify which leakages are important.

2.4 Risks in Authorization Context

One challenge in access control enforcement is to restrict subjects to appropriate permissions for the individual requests they make. Typically, access control treats each subject uniformly for all requests. For example, the operating system grants each subject the same set of permissions for any request made.

Such a "black box" approach can lead to two kinds of problems. First, an adversary may try to attack the system using their available permissions, and we may want to limit the adversary to a subset of their permissions based on some contextual knowledge related to the request. For example, ContexIoT [29] aims to limit the permissions available to processes on IoT devices based on the control flow and data flow that led to the request being submitted. In this case, the authors instrument untrusted programs to gather control and data flows that are used in access control enforcement. Second, victims may be protected from compromise if access control enforcement limits the permissions they may use when processing their requests. For example, Jigsaw [47] limits the permissions available to a process when it opens a file to prevent *confused deputy* attacks [26, 40]. If a victim uses input from an adversary to build a file name, the victim is restricted to only access objects available to untrusted parties. Again, knowledge of data flows (e.g., to constructing file names) are used to restrict access.

We find that risk occurs because subjects may have too many permissions available for a particular context. Risk of authorization context occurs because we do not know enough about the program context to reduce the subject's permissions. For example, without program context, we cannot identify the two cases above, where subjects may maliciously or accidentally misuse their permissions.

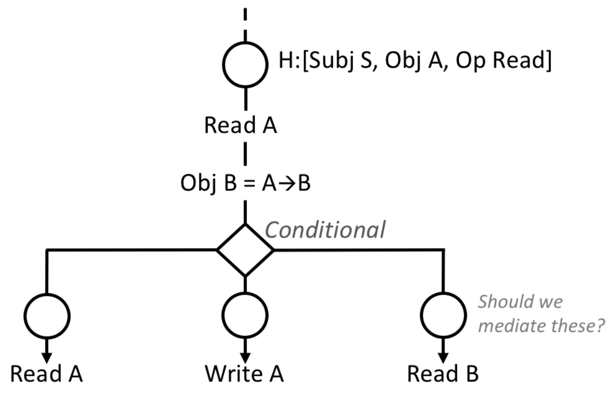

Figure 2: Risks in placing authorization hooks that do not mediate every security-sensitive operation

2.5 Risks in Mediation Granularity

Finally, the assumption that the authorization system mediates all the security-sensitive operations correctly may be flawed, leading to additional risks. A major concern is that an authorization system does not satisfy the *complete mediation* property of the reference monitor concept [6]. When one considers failures in complete mediation, one normally thinks of a security-sensitive operation that may not be mediated by any authorization hook in some execution trace. Such errors have occurred in the deployment of authorization systems [17, 46], but some operations may not require mediation and even mediated operations may be incorrect.

In particular, we find that risks occur because programmers may understand whether operations may allow unauthorized access. Even when an operation is mediated by an authorization hook, this problem may occur for two reasons. First, programmers aim to minimize the number of authorization hooks (e.g., for performance or to keep the policy simpler), so they may only mediate the first security-sensitive operation and assume subsequent operations are protected. Figure 2 shows an example. Suppose that an authorization hook checks whether subjects can read an object A. However, subsequent statements may write to that object or may operate on different objects, such as object B extracted from a field of object A. In general, no subsequent authorization hook is necessary as long as all the subjects authorized for the mediated operation (e.g., read of the object in the example) are also authorized to perform all security-sensitive operations that may directly dominated by that hook [35]. In the example, should some of the authorized subjects for reading object A not be allowed to write object A or not be allowed to access all the objects that may be assigned to object B, then other authorization hooks would be necessary to block those unauthorized accesses while allowing the read operation.

Second, mediation may only be intended to allow limited access to an object. Consider a database where some fields contain secret data. We may authorize a subject to access a record, and even allow updates to a secret field based on such accesses. However, we would want to prevent the values in that secret field from being released publicly (e.g., sent over the network) [4]. Typically, programmers add authorization hooks and sanitizers independently, meaning that there may be a risk of a missing sanitizer even after authorization. In addition, for integrity, Amir-Mohammadian and Skalka [5] prevent victims from using adversary-controlled data without sanitization to augment access control.

3 SECURITY MODEL

In this work, we assume that programmers and system administrators are benign. They may make mistakes in the configuration of access control policies (administrators) or in the enforcement of access control policies (programmers), but they are not active adversaries in the system. We further assume that the systems upon which programs enforcing access control run are able to prevent compromise.

On the other hand, all the processes that make access control requests may be compromised. Some processes may aim to escalate their current privileges by trying to compromise another process with greater access. Others may simply want to exploit permissions already assigned to them (e.g., insiders).

4 RISK MODEL OBJECTIVES

In this paper, we explore the requirement for a system that reasons about the risk incurred during access control enforcement, which covers the four sources of risk described in the problem section: (1) risk due to authorizing unsafe operations; (2) risk due to abuse of authorized permissions; (3) risk due to lack of context; and (4) risk due to the granularity of authorization hooks. To do so, we have to overcome several challenges. First, we aim to capture *real* risks. Static analysis shows possible risks, but does not tell us whether risks actually appear. Second, we aim for risk to be *monotonic* [39]. That is, we aim to identify risks in access control enforcement that add attack options for adversaries, such that the risk value computed must be greater if the risk was taken than if it was not taken. Third, we envision that risky behaviors must be *proportional* to its impact. For example, an action that does not enable any privilege escalation must create a lower risk than an action that does enable privilege escalation. Ideally, we could compute a risk cost for each operation and determine whether that cost is within a risk budget, analogous to differential privacy [16]. Unfortunately, we presently lack the formal foundations of differential privacy in evaluating access control, but perhaps studying how risk may accumulate may lead to insights in the future.

To capture real risks, we propose to compute risk by tracking the dynamic behaviors in program executions. To capture unsafe operations and abuse of authorized permissions, we propose to collect risks associated with these operations as they are authorized. To capture risks from security-sensitive operations that are not explicitly authorized, we collect risks associated with those operations when they are run.

To capture risks monotonically, we propose to separating the accumulation of threats from the risks actually taken by targets. As shown in Figure 1, there is both a threatening operation (e.g., writing to an object used by a high integrity party) and a risky operation (e.g., reading the object by the high integrity party). A *threatening operation* creates a potential for risk by propagating threats from untrusted subjects to objects. The potential risk created by threatening operations accumulates in objects, which we call *object risk*. A *risky operation* occurs when a target subject accesses (reads or executes) an object that has been threatened, transferring the risk from the object to the target subjects. That is, a threat is not truly a risk until at least one target subject performs a risky operation on a threatened object. Thus, if an object if threatened, but the risk never is taken (e.g., the object is deleted before being accessed), then the threats are not turned into risk for any subject. However, if a risk is taken by a subject, that risk is maintained with the subject even if the threatened object is removed later (e.g., to hide the attack).

To capture risks proportionally, we propose to estimate the value of the risk based on how the risk may impact the victim. For example, we explore estimating risk based on three factors: (1) *the scope of the threat created by adversaries*; (2) *how unique the risk is*; and (3) *how much an adversary may gain from the risk.* First, an adversary may create a larger risk by controlling more data used by the victim. Also, for each threat, we increase the potential for risk based on the uniqueness of the unsafe operation. For example, if very few low integrity subjects can write to a particular object, then we envision that the potential for risk of this subject writing to the object is greater than if many subjects may have the same permission. Third, we assume that a threat that may enable an adversary to gain access to more new privileges should be a higher risk than one that only grants fewer new privileges.

Finally, to compute risk, we propose to collect risk estimate information at runtime. We find that access control enforcement allows many risks to be taken, but if such risks are expected (e.g., receiving a network packet) and infrequent then the risk estimate for the subject should be low. However, modifying programs in an ad hoc manner is complex and error-prone. We envision that program code to collect information for risk estimates at runtime should be generated from declarative specifications, as proposed for auditing code [4]. One concern is the overhead incurred by runtime logging, so we suggest exploring optimizations to compute information statically, such as proposed for ContexIoT [29].

5 RUNTIME RISK ESTIMATION

5.1 Computing Risk Estimates

The idea of computing risk is to record for each threatening operation how much risk is created by the threat. The risk created by threatening operations is accumulated at the object for each threatening operation. Then, when a target subject performs a risky operation, the risk transfers from the object to the subject that invoked the risky operation. A subject's risk then is an accumulation of the risks collected from the threatened objects that that subject accesses.

In theory, if an object utilized (i.e., read or executed when considering integrity) by a subject is fully controlled by an adversary, then the risk faced by that subject is *complete* for that object. However, a complete risk may or may not provide an advantage to an adversary, depending on what an adversary gains from the risk. For integrity violations, we consider the privilege escalation that an adversary may gain from the permissions available to the subject, such as by compromising the subject completely or exploiting its permissions as a confused deputy [26, 40].

Thus, we envision a risk estimation method where a complete risk incurs a risk estimate of 1 and no risk incurs a risk estimate of

0. Given information flow as a motivation, a risk estimate of 0 for a subject would occur if the subject only reads objects that have never had a threatening operation. On the other extreme, a subject that only performs risky operations on objects whose data is fully threatened would incur a risk value of k, where k is the number of objects read and each object has a risk value of 1. To estimate risk between these two extremes, we propose equations that relate the adversary control of an object to the risk of accessing it. The equations presented are strawmen, so the focus should be on the elements that constitute risks.

To capture the proportionality of risk, we modulate the risk estimates by the uniqueness of the threatening operation and the advantage to be gained by the adversary through that operation. First, if a risk can rarely be created, i.e., very few subjects could perform the threatening operation, then we propose that the risk estimate should be increased. Second, if a risk may have a big payout, i.e., enables an untrusted subject to gain access to many or critical privileged permissions, we also propose to increase the risk estimate based on the fraction of privileged permissions that are available to the target subject performing a risky operation. In this paper, we say that *privileged permissions* are permissions not available to any low integrity subject.

Example: Risk Flows between Subjects and Objects. We now demonstrate the proposed approach for risk estimation with a simple example based on Figure 1. Suppose that (low integrity) Subject A contributes 10 bytes to Object X, which consists of 100 bytes altogether. If the remaining 90 bytes are from writes from high integrity subjects, then one could say that Object X's risk estimate is 0.1 (10 bytes out of 100 bytes). When (high integrity) Subject Y reads from Object X, the risk estimate of Subject Y is updated based on the risk estimate of Object X. Suppose that Subject Y only reads or executes Object X, then the risk estimate of Subject Y will be the value of Object X at the time of read (risky) operation, which could be 0.1. We will examine the specific approaches proposed for computing risk estimates for objects and subjects in the remainder of this section.

5.2 Risk Estimation for Unsafe Operations

Unsafe operations are authorized by the existing authorization hooks, so we can update risk estimates when such unsafe operations are authorized. The main challenge is to determine whether an operation is unsafe, and hence either a threatening or risky operation. To do this, the system must have identified the low integrity (high secrecy) and high integrity (low secrecy) subjects in the access control policy. For example, researchers have used knowledge of how system's boot [28] or which subjects may attack the kernel [49] to estimate low and high integrity subjects. Identifying high secrecy subjects is more difficult because a wider variety of subjects and objects contain secret information. We assume that some choice of trusted and untrusted subjects has been made a priori and utilize this knowledge to identify unsafe operations.

However, a second challenge occurs if other threatening operations change the risk estimate of an object while it is in use. Suppose that a high integrity subject and a low integrity subject have a file open concurrently. Thus, if the low integrity subject continues to write the object after the first time the object is read by the high integrity subject, we may compute the risk estimate incorrectly if we only update risk when the `open` operation occurs. Although authorization may occur at the `open` operation, we need to be able to update the risk estimate each time an object is used (e.g., `read` operations).

To compute a risk estimate for unsafe operations, we propose that the risk estimate account for: (1) the *scope* of the threat; (2) the *uniqueness* of the threat; and (3) the *gain* for the adversary in terms of the privilege escalation possible via compromise. For a threatening operation, the scope of the threat may be reflected by the amount of threatening data contained in the object,

$$scope(obj) = x(obj)/d(obj) \tag{1}$$

where: (1) $x(obj)$ is the amount of threatening data (i.e., data written by threatening operations) written to the object and $d(obj)$ is the total amount of data for the object. Presumably, the more adversary-controlled data that a program must process, the greater the risk, although the risk may not be linear as reflected here.

However, the threatening operation may be common, which should reduce the threat of any individual operation, as the more common a threat, the more likely it is to be addressed. Thus, we may want to adjust the risk caused by an unsafe operation in an inverse relationship to the number of subjects who may perform it,

$$uniqueness(obj) = f(potentialThreat(obj)) \tag{2}$$

where f is a function and $potentialThreat(obj)$ is the fraction of threatening subjects that are authorized to write to the object. This term aims to adjust the risk estimate based on the uniqueness of the adversary's ability to create the threat from $f(potentialThreat(obj)) = 0$ (if common) and approximately $f(potentialThreat(obj)) = 1$ (if rare). f could be linear (i.e., $1 - potentialThreat(obj)$), but alternatives may reflect that risks emerge mainly from rare permissions (higher order) or that risk only reduces if a permission is very common. We will explore these alternatives.

Thus, for each object its associated risk is estimated by,

$$risk_{un}(obj) = scope(obj) \times uniqueness(obj) \tag{3}$$

For a subject, we want to produce a risk estimate that reflects the impact of the risky unsafe operations that the subject has performed. For example, a subject's risk estimate may depend on the combination of the risk estimates for the objects that the subject has accessed,

$$risk_{un}(subj) = \sum_{i=1}^{n} max(risk_{un}(obj_i)) \tag{4}$$

where n is the number of objects accessed by the subject. The maximum value is used here for each object to account for the fact that the risk associated with an object may change at each access of the object by the subject. Thus, by using the max we account for the access with the higher risk. An alternative would be to have n be the number of operations, where risk would be summed using the current object risk value. However, that approach may enable an adversary to hide risk, if an object is accessed many times in a safe form.

Finally, the risk to a subject should also factor in the benefit of exploiting the permissions available to the subject. Thus, we suggest multiplying the risk estimate resulting from risky operations by the privilege escalation possible via compromise *potentialGain*(*subj*),

$$risk_{un}(subj) = \sum_{i=1}^{n} max(risk_{un}(obj_i)) \times potentialGain(subj) \quad (5)$$

where *potentialGain*(*subj*) is the fraction of privileged permissions accessible to the subject. The fraction of the privileged permissions available to a subject indicates the amount of privilege escalation that may be achieved by an adversary, which justifies an adversary trying to exploit the subject. Privileged permissions may include high secrecy and high integrity permissions for resources that must be protected in the system.

These risk equations leave open the question of which bytes in a file currently may have originated from a threatening operation or a safe operation. The problem is that both threatening and safe data may be overwritten, so all the prior threatening (or safe) data may have been removed or at least reduced. Tracking file taint per byte may be expensive, but simply tracking the amount of data written can be straightforward since we can simply log these operations. We will conservatively assume that all the threatening data written remains in the file, and explore this problem further in future work.

5.3 Risk Estimation for Permission Abuse

In this section, the challenge is to map the threatening and risky operations to operations performed by a subject with their own permissions. In this case, the threat is due to sensitive data defined by or protected by a high integrity subject that may be tampered with or leaked should that subject be compromised or be a malicious insider. Thus, we must identify threatening and risky operations, and determine how to compute risk estimates for each.

In this case, we view threats as being the operations that cause data to be sensitive. If some subject adds sensitive data to an object that can be accessed by a target subject, then we want to measure how much of this sensitive data each target subject may put at risk by accessing those objects. However, programmers do not explicitly identify sensitive information, so we need a way to detect operations that cause an object to become sensitive. A simplistic approach is to assume that all the data of each object available to a subject is sensitive. We can mitigate the impact of the threat by determining if adding data to the resource is a common operation. If so, it is likely the data is less sensitive.

As a result, the risk estimate for objects related to permission abuse is calculated as follows,

$$risk_{ab}(obj) = scope(obj) \times actualThreat(obj) \quad (6)$$

where: (1) *scope*(*obj*) is again the fraction of sensitive data written in the object (which may be assumed to be 1 in some cases) and (2) *actualThreat*(*obj*) estimates the sensitivity of the object based on the fraction of subjects that wrote (read) the object for integrity (for secrecy). We plan to track the amount of data produced by threatening operations x as described above, but the challenge is to estimate which operations produce sensitive information. To do this we propose to leverage the commonality of access to the data, where a higher commonality reduces the risk of permission abuse. For integrity, the higher the fraction of subjects that can write to the object, then the lower the sensitivity of the object. Since few objects may be written (read) by all subjects, the estimate for *actualThreat*(*obj*) may be normalized to fill more of the range between 0 and 1 (e.g., use the maximum fraction of subjects).

Risky operations occur when a subject uses its permissions to access a threatened object. We propose that the risk estimate for subjects abusing permissions is calculated as follows,

$$risk_{ab}(subj) = \sum_{i=1}^{n} max(risk_{ab}(obj_i)) \times actualGain(subj) \quad (7)$$

where $max(risk_{ab}(obj_i))$ is the maximum value of object i's risk estimate across all accesses so far by the subject upon this object. The function *actualGain*(*subj*) adjusts the risk sum by the fraction of sensitive permissions that the subject has utilized. The *actualGain* term captures the level of abuse across all the sensitive data in a system.

5.4 Risk Estimation for Authorization Context

Other risks occur because of the difficulty in predicting which permissions to authorize in which program execution contexts. First, malicious programs may abuse the permissions that they are granted and request permissions that are not relevant to the execution context to spy on users or leak information. Second, programs may fall victim to attacks that trick them into using their privileged permissions on behalf of adversaries, in the so-called *confused deputy attacks* [26, 40]. Finally, the party performing authorization (e.g., operating system or server program) typically does not track the execution context of a program requesting permissions. Thus, traditional access control does not identify such abuses.

In general, risks of malicious programs or confused deputy attacks because access control enforcement does not consider program context occur because the program may access different objects through the same system call invocation. Researchers have found that while programs may access lots of different objects, several system calls only access a small number of objects and only access objects with similar properties. Vijayakumar [48] found that 67% of the statements that invoked system calls to retrieve an object (e.g., calls to the libc function `open`) retrieve only one object in the programs in a LAMP stack (i.e., Apache, MySQL, and PHP). 78% of the statements retrieved objects of only one SELinux label. Thus, risk estimate due to the objects retrieved $risk_{re}(subj)$ should be based on the variance in security properties of objects retrieved,

$$risk_{re}(subj) = \sum_{k=1}^{m} classes(subj, stmt_k) \times actualGain(subj) \quad (8)$$

where m is the number of system call statements and *classes* is the fraction of classes of objects associated with a particular statement that retrieves objects for that subject ($stmt_k$). Classes of objects can be based on the object name (one or more), label (one or more), secrecy and integrity classification. In general, the worst case occurs when a subject has a single statement that may retrieve a variety of objects that differ in their security classifications (e.g., high and low integrity as well as high and low secrecy). In such a case, the program has to determine how to enforce a variety of security requirements itself, creating more risk.

Another aspect of risk is the way that the program gathers the information that leads to system calls that retrieve or access system objects. Researchers have explored methods that collect control flows and data flows, as well as detect particular operations, such as filtering, to judge whether to allow a program to perform a particular system call [29, 47]. For example, the complexity of the control and data flows that lead to a particular system call statement in terms of the size of these flows may be indicative of risk,

$$risk_{flow}(subj) = \sum_{k=1}^{m} flow(subj, stmt_k) \times actualGain(subj) \quad (9)$$

where $flow$ computes a risk estimate based on the complexity of the control and data flows that produced the inputs to the statement. The complexity should be normalized (between 0 and 1) to be consistent with other risk values. Computing risk estimates from flows will require the application of static analysis tools to collect such information at runtime. Thus, we will defer calculation of such risk to future work.

5.5 Risk Estimation for Mediation Granularity

Recall that risks due to mediation granularity are caused because a programmer chooses not to directly mediate some security-sensitive operations. We find that computing risk estimates related to mediation granularity is essentially the same as computing the risk estimate for unsafe operations, as described above. The difference is that the administrators do not specify policy for unmediated operations, so the main challenges are to identify which unmediated security-sensitive operations are unsafe and how many untrusted subjects can perform threatening operations. In addition, identifying security-sensitive operations is itself an imprecise process that induces risk.

An operation is unsafe if it causes an illegal information flow as shown in Figure 1. Threatening operations occur when untrusted subjects actually access objects that may be accessible to trusted subjects. Although we lack a policy to determine whether a trusted subject may be able to access an unmediated object explicitly, we can record risk estimates for all unmediated objects accessed by untrusted subjects. We envision that programs would record risk estimates for objects accessed in unsafe, unmediated security-sensitive operations in the same way as described in Section 5.2.

Risky operations transfer the objects risk estimate to the subject that has taken the risk, as for unsafe operations. One question is how to estimate the impact of privilege escalation given that we are tracking objects for which there is no explicit authorization policy. One simple approach would be to simply use the privilege escalation possible in the specified access control policy alone, although this misses the possible escalation in unmediated security-sensitive operations. Researchers need to develop methods to extrapolate policies for unmediated operations to estimate where escalation is possible.

Finally, as mentioned in Section 2.1, identifying security-sensitive operations itself induces risk. First, both data-flow and control-flow methods require programmers to manually identify untrusted and secret inputs, which itself is error-prone [34, 45]. Second, the computation of security-sensitive operations using control flow involves heuristics and the resolution of information-flow violations using data flow involves possible imperfect sanitizers. Again, we must log information necessary to compute risk estimates and determine how such inaccuracies may impact risk estimation.

We propose to estimate the risk for mediation granularity as follows,

$$risk_{me}(subj) = \sum_{i=1}^{n} max(risk_{un}(obj_i)) \times SSO_{unmed}(subj) \quad (10)$$

where $SSO_{unmed}(subj)$ is the fraction of unmediated security-sensitive operations (SSOs). We identify three types of unmediated SSOs: (1) "open" SSOs that are not control-flow dominated by any authorization hook on any control-flow path leading to those SSOs; (2) "partial" SSOs that are control-flow dominated by an authorization hook on some, but not all, control-flow paths leading to those SSOs; and (3) "dominated" SSOs that are control-flow dominated along all control-flow paths leading to those SSOs. We consider the dominated SSOs to be the least risky, although because these operations are not directly associated with an authorization hook they may still incur risk. Interestingly, we consider "open" SSOs to be less risky than "partial" SSOs, because an open SSO may legitimately be accessible to all subjects or not actually access security-sensitive resources. On the other hand, a "partial" SSO is expected to be mediated along some paths. Thus, we propose to compute $SSO_{unmed}(subj)$ as a combination of the three types of unmediated operations, where their fractions are augmented by multiplicative factors α, β and γ, for dominated, open, and partial, respectively. In addition, we require that $\alpha < \beta < \gamma$ to account for their relative levels of risk.

6 EXPERIENCES WITH RISK

In this section, we examine experimental results in computing risk estimates for access control enforcement. Primarily, we evaluate use of the risk estimate equations proposed in Section 5 on Android systems. The main question that we aim to answer is whether the inputs to risk estimation described above may actually be computed in practice and what the inputs currently look like for benign system operation.

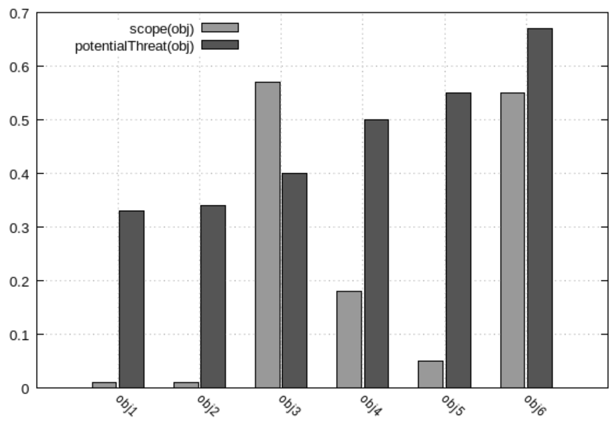

Figure 3: Scope and fraction of threatening subjects (potential threat) for the six untrusted objects accessed by the `mediaserver`

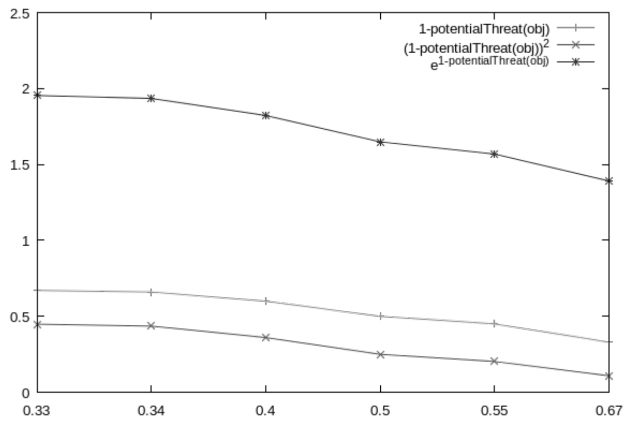

Figure 4: Uniqueness values for the six untrusted objects accessed by the `mediaserver`, plotted for three different choices of the f function: $f_1(obj) = 1 - potentialThreat(obj)$, $f_2(obj) = e^{1-potentialThreat(obj)}$, **and** $f_3(obj) = (1-potentialThreat(obj))^2$

6.1 Experiences with Unsafe Operations

Regarding tracking unsafe operations, we examined a stock Android policy for the Android 6.0.1 (kernel 3.4.0) version. To determine how many unsafe operations are performed, we utilize the Android Compatibility Test Suite (CTS) [2]. CTS is designed to reveal functional incompatibilities between applications and the Android system. CTS runs unit tests to test for incompatibilities. Using CTS, we ran 127,058 tests over 20 hours and 44 minutes. The access control policy is enforced by SEAndroid.

What we see in Android is that for 1,264,978 operations there are 926,491 operations that are threatening operations, but only 445 operations that are risky operations. While this may appear surprising, CTS creates a factor of 10 more write-like operations than read-like operations. That is, while the untrusted subjects may perform a wide variety of operations that may threaten privileged, high-integrity subjects, the high-integrity subjects only use risk operations for a very small fraction of their authorized operations. Thus, most of the risk estimate computation effort would apply to threatening operations. Future risk estimation methods should focus on low-overhead for threatening operations and put more effort for assessing risky operations.

Furthermore, we examined a macrobenchmark of 15 system apps shipped with the stock Android operating system for the Android 6.0.1 (kernel 3.4.0) version. We used the Android UI/Application Exerciser Monkey [1] to target sensitive operations and automatically generate sequences of input events.

From the data collected via the UI/Application Exerciser we calculated the risk taken by 10 Android processes selected among those implementing core features of the operating system. The selected processes are listed in column 1 of Table 1, whereas the corresponding calculated risk for unsafe operations ($risk_{un}$) is listed in column 5 of Table 1.

To calculate such values of risk we adopted equation (5). In particular, we started by measuring the scope of all untrusted object accessed by each process as per equation (1), and the fraction of subjects that are authorized to write to such object. For example, Figure 3 plots the measured values for the 6 untrusted objects accessed by the `mediaserver` process, as reported in column 2 of Table 1. As one can see, adversaries often control a small amount of data in threatened objects (scope), but few enough subjects can exercise the threat to make that estimate significant (potentialThreat).

For our experimental measurements, we have considered as untrusted objects all files that can be written by low-integrity processes, which were identified as the complement of the transitive closure of transition rules for subject types starting with the `kernel` label. Such choice is motivated by the fact that after the kernel is loaded during the boot process, the initial process is assigned the predefined initial SELinux ID `kernel`, which is used for bootstrapping before the policy is loaded.

We then proceeded with the calculation of the uniqueness value, which we used to calculate the risk reported in column 5 of Table 1, for such objects by adopting equation (2). Figure 4 plots the measured values for the 6 untrusted objects accessed by the `mediaserver` process by considering three alternative choices for the f function to model a linear, a polynomial, and an exponential dependency of the uniqueness factor from the potential threat.

6.2 Experiences with Permission Abuse

As shown in Table 2, using CTS again over the stock Android policy we found that no trusted subject uses more than 143 of their permissions (su) or greater than 12% of their permissions (system_server). As a result, permission use greater than about 20% may be considered unusual. Thus, we may want to adjust the risk estimation approach from a linear approach to one that accelerates risk at lower fractions. Alternatively, we may compare permission use to an average over a period of time.

Furthermore, from the data collected via the UI/Application Exerciser, we calculated the risk taken by 10 Android processes when considering possible permission abuse ($risk_{ab}$) as listed in column 6 of Table 1. To calculate such values of risk we adopted equation (7). In particular, we started by measuring the fraction

Table 1: Risk measurements for the ten core Android processes. We report the untrusted objects and the unmediated security sensitive operations (SSOs) as fractions of the total number of objects and total number of SSOs respectively

Subject	Untrusted Objects	Call Sites	Unmediated SSOs	$risk_{un}$	$risk_{ab}$	$risk_{re}$	$risk_{me}$
mediaserver	6/115	6	17/334	0.0246	0.0031	0.0626	0.0417
init	2/96	18	5/145	0.03	0.0164	0.0299	0.0259
main	8/1,122	8	18/2,511	0.0234	0.035	0.0039	0.0028
activitymanager	9/7,650	11	45/27,552	0.012	0.0057	0.0013	0.0005
servicemanager	1/568	8	2/1,018	0.0105	0.0053	0.0005	0.0007
surfaceflinger	1/5	4	4/55	0.0019	0.0002	0.008	0.0705
keystore	1/24	5	8/129	0.0077	0.0019	0.002	0.0477
netd	8/48	10	7/130	0.0176	0.0039	0.0667	0.0473
rild	1/3	2	1/10	0.0297	0.0002	0.03	0.099
sdcard	1/104	8	4/409	0.008	0.0098	0.0023	0.0039

Table 2: Permissions usage information for a run of the Android Compatibility Test Suite for the top ten subjects (by number of permissions)

Subject	Used Permissions	Total Permissions	Fract. of Used Permissions
su	143	1,983	0.07
dumpstate	96	1,237	0.08
system_server	92	787	0.12
system_app	26	468	0.06
radio	18	531	0.03
nfc	16	406	0.04
bluetooth	15	497	0.03
mediaserver	14	655	0.02
shell	5	1,272	< 0.01
surfaceflinger	5	329	0.01

of sensitive data in each object $scope(obj)$ again calculated as the amount of data written by low-integrity subjects to the object. We then measured the sensitivity of the object $sensitivity(obj)$ as fraction of subjects that can write the object. By following equation (6) we obtained the risk estimate for each object related to permission abuse. We then used such estimate to calculate the corresponding risk for each of the subjects accessing the object by following equation (7).

6.3 Experiences with Authorization Context

The data collected via the UI/Application Exerciser also allowed us to calculate the risk taken by the 10 Android processes when considering their authorization context while retrieving objects ($risk_{re}$). The measurements are listed in column 7 of Table 1. To calculate such values of risk we adopted equation (8). In particular, we identified all the call sites (or system call statements) used by each of the 10 processes when accessing untrusted objects. We enumerated them in column 3 of Table 1. We then calculated $classes(subj, stmt_k)$ as 1 plus the fraction of trusted object labels times the fraction of untrusted object labels (over the total objects labels specified in the policy) seen at call site k when retrieving untrusted objects. This is in line with the intuition that the risk increases if the subject access a higher number of untrusted objects, and increases even further if the subject also accesses a set of trusted objects.

Table 3: Evolution in authorization hook counts for X Windows. Manually placed hooks are compared to "Automated Hooks" generated using a proposed method [34]

XServer versions	Automated Hooks	Manual Hooks
1.7	278	186
1.8	280	186
1.9	276	181
1.10	272	180
1.11	284	180
1.12	312	207
1.13	333	206
1.14	333	206
1.15	317	207
1.16	310	207
1.17	334	207

Also, our findings are in line with previous research [29] reporting that the number of operations requested per application in a particular context on average is only 3.5 with no more than 30 contexts being seen. Again, this runtime analysis may be prone to false positives. However, one can see that while restricting permissions available in individual contexts may lead to program failures (due to blocking necessary permissions), risk may be limited to only a small number operations because only a few program contexts require risk and only a small fraction of the program contexts are complex and/or variable.

6.4 Experiences with Mediation Granularity

As shown in Table 3, developers often have difficulty determining the correct placement of authorization hooks for their programs. As can be seen, the actual number of authorization hooks in X Windows varied slightly with each version, excepting between versions 1.11 and 1.12 where a significant change was made to the program. Also, we compare the number of hooks placed manually to the number of security-sensitive operations computed using a control-flow method [35]. As one can see, there is not a clear correlation between changes in the number of hooks placed manually and changes in the number of security-sensitive operations identified automatically.

Additionally, we calculated the risk taken by the ten core Android processes when considering their mediation granularity in security sensitive operations, shown as the $risk_{me}$ column in Table 1. We define the security-sensitive operations in this study as those operations that operate on data types considered security sensitive, such as files, sockets, and databases. Using data types to identify security-sensitive operations has been a common approach [24].

In our study, such operations are said to be "unmediated" if they were not preceded by a permission check (either SELinux or Android permission check) in the most enclosing control block. We then classify the unmediated operations using a static control-flow analysis to detect whether the operation is mediated along all control-flow paths (dominated), no control-flow paths (open), and some control-flow paths (partial). We set the risk factor of each type, α, β, and γ to the values 1, 2, and 3, respectively, for application of equation (10).

7 DISCUSSION

We note that the risks computed in this evaluation are examples of "baseline" risks, not risks under attack conditions. Programs are run in a benign manner, so the risk values computed are modest in many cases. None of the programs evaluated in Table 1 incur a risk estimate of over 0.1 for any of the four categories.

An aim of using risk estimates would be to use accumulated risk to motivate changes in security posture, analogous to using accumulated privacy cost to block queries in differential privacy. However, at present, we lack a definition of security risk that precisely describes a security property as meaningful as differential privacy. We advocate comparing accumulated risk of one system or program to that system or program under conditionals that comply with safe, expected use to identify when risk estimates differ by more than an expected amount as a basis for adjusting security decision-making. By identifying dimensions upon which risk may be tracked and proposing strawman risk estimates that accumulate, we hope to further investigate how risk estimates may relate to security properties in the future.

Finally, the focus of this research should primarily be on the types of risk related to authorization, one type of security mechanism. We propose simple equations for each type of risk, but much more work will be necessary to devise equations that capture expected security properties.

8 CONCLUSIONS

In this paper, we present a study of risk in access control enforcement. While we have long had principles describing how access control enforcement should be implemented, imprecision in access control mechanisms and access control policies leads to risks that may enable exploitation. We identify four types of risk in access control enforcement, two that relate to access control policies and two that relate to access control mechanisms. We propose an approach for estimating risk as subjects perform operations. At a high level, the approach aims to leverage tracking of information directed at risk estimation and the development of risk estimation approaches that are monotonic and account for the impact of risks proportionally. We examine challenges in implementation such risk estimation for the four types of risks. Our evaluation shows that in normal use programs due not take excessive risks for the programs we investigated directly under benign use conditions. Thus, it may be possible to use such risk estimation to gauge when defenses should be added (e.g., more authorization hooks) and/or operations should be blocked or audited more carefully (e.g., tighter access control policies) when seeing an excessive or increased accumulation of risk. We propose using the scope, uniqueness, and gain possible via risk as principles underlying risk estimation, although a formal definition of a risk property for access control remains future work.

ACKNOWLEDGEMENTS

This research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

This research is also based in part upon work supported by the National Science Foundation (NSF) under Grant Numbers CNS-1408880 and CNS-1408801. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Android UI/Application Exerciser. https://developer.android.com/studio/test/monkey.html.
[2] Compatibility Test Suite — Android Open Source Project. https://source.android.com/compatibility/cts/.
[3] Tresys. SETools - Policy Anakysis Tools for SELinux. https://github.com/TresysTechnology/setools3/wiki.
[4] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. 2016. Correct Audit Logging: Theory and Practice. In *Principles of Security and Trust (POST).*
[5] Sepehr Amir-Mohammadian and Christian Skalka. 2016. In-Depth Enforcement of Dynamic Integrity Taint Analysis. In *ACM Programming Languages and Security Workshop (PLAS).*
[6] J. P. Anderson. 1972. *Computer Security Technology Planning Study, Volume II.* Technical Report ESD-TR-73-51. Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA.
[7] D. E. Bell and L. J. LaPadula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation.* Technical Report ESD-TR-75-306. Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA.
[8] K. J. Biba. 1977. *Integrity Considerations for Secure Computer Systems.* Technical Report MTR-3153. MITRE.
[9] Khalid Zaman Bijon, Ram Krishnan, and Ravi S. Sandhu. 2013. A framework for risk-aware role based access control. In *IEEE Conference on Communications and Network Security.* 462–469.
[10] Hong Chen, Ninghui Li, and Ziqing Mao. 2009. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS).*
[11] Liang Chen and Jason Crampton. 2011. Risk-Aware Role-Based Access Control. In *Proceedings of 7th International Workshop on Security and Trust Management.* 140–156.

[12] Ying Chen, Heng Xu, Yilu Zhou, and Sencun Zhu. 2013. Is This App Safe for Children?: A Comparison Study of Maturity Ratings on Android and iOS Applications. In *Proceedings of the 22nd International Conference on World Wide Web*. 201–212.

[13] Pau-Chen Cheng, Pankaj Rohatgi, Claudia Keser, Paul A. Karger, Grant M. Wagner, and Angela Schuett Reninger. 2007. Fuzzy Multi-Level Security: An Experiment on Quantified Risk-Adaptive Access Control.. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. 222–230.

[14] Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. 2012. Is This App Safe?: A Large Scale Study on Application Permissions and Risk Signals. In *Proceedings of the 21st International Conference on World Wide Web*. 311–320.

[15] D. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–242.

[16] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. 9, 3 (2014), 211–407.

[17] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. 2002. Runtime verification of authorization hook placement for the Linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 225–234.

[18] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA, 235–245. DOI: http://dx.doi.org/10.1145/1653662.1653691

[19] W. Enck *et al.* 2010. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*.

[20] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. 627–638.

[21] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *IEEE Symposium on Security and Privacy*. 636–654.

[22] V. Ganapathy, T. Jaeger, and S. Jha. 2005. Automatic placement of authorization hooks in the Linux security modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. Alexandria, VA, USA.

[23] V. Ganapathy, T. Jaeger, and S. Jha. 2006. Retrofitting Legacy Code for Authorization Policy Enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. To Appear.

[24] Vinod Ganapathy, David H. King, Trent Jaeger, and Somesh Jha. 2007. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 38th International Conference on Software Engineering*. 458–467.

[25] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire. In *Proceedings of the 20th USENIX Security Symposium*.

[26] N. Hardy. 1988. The Confused Deputy. *Operating Systems Review* 22 (1988), 36–38.

[27] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. 2007. From trusted to secure: building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 1–14.

[28] T. Jaeger, R. Sailer, and X. Zhang. 2003. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the 12th USENIX Security Symposium*. 59–74.

[29] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'17)*.

[30] Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu. 2014. RiskMon: Continuous and Automated Risk Assessment of Mobile Applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. 99–110.

[31] Savith Kandala, Ravi S. Sandhu, and Venkata Bhamidipati. 2011. An Attribute Based Framework for Risk-Adaptive Access Control Models. In *Proceedings of the Sixth International Conference on Availability, Reliability and Security*. 236–241.

[32] Stephen McCamant and Michael D. Ernst. 2008. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*. Tucson, AZ, USA, 193–205.

[33] Frank D. McSherry. 2009. Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. 19–30.

[34] Divya Muthukumaran, Trent Jaeger, and Vinod Ganapathy. 2012. Leveraging "Choice" to Automate Authorization Hook Placement. In *Proceedings of the 19^{th} ACM Conference on Computer and Communications Security*. ACM Press, Raleigh, North Carolina, USA.

[35] Divya Muthukumaran, Nirupama Talele, Trent Jaeger, and Gang Tan. 2015. Producing Hook Placements to Enforce Expected Access Control Policies. In *Proceedings of the 2015 International Symposium on Engineering Secure Software and Systems*.

[36] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. *ACM Operating Systems Review* 31, 5 (Oct. 1997), 129–142. http://www.cs.cornell.edu/andru/papers/iflow-sosp97/paper.html

[37] Qun Ni, Elisa Bertino, and Jorge Lobo. 2010. Risk-based Access Control Systems Built on Fuzzy Inferences. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. 250–260.

[38] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Security Symposium*. 527–542.

[39] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. 2012. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 241–252.

[40] Giuseppe Petracca, Yuqiong Sun, Trent Jaeger, and Ahmad Atamli. 2015. Audroid: Preventing attacks on audio channels in mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 181–190.

[41] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and Privacy for MapReduce. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*.

[42] Farzad Salim, Jason Reid, Ed Dawson, and Uwe Dulleck. 2011. An Approach to Access Control under Uncertainty. In *Proceedings of the Sixth International Conference on Availability, Reliability and Security*. 1–8.

[43] J. H. Saltzer *et al.* 1975. The Protection of Information in Computer Systems. *Proc. IEEE* (1975).

[44] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2011. RoleCast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*.

[45] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing Access-Control Bugs in Web Applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*.

[46] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: automatically inferring security specifications and detecting violations. In *USENIX Security*.

[47] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. 2014. JIGSAW: Protecting Resource Access by Inferring Programmer Expectations. In *Proceedings of the 23rd USENIX Security Symposium*.

[48] Hayawardh Vijayakumar and Trent Jaeger. 2012. The Right Files at the Right Time. In *Proceedings of the 5th IEEE Symposium on Configuration Analytics and Automation (SafeConfig 2012)*.

[49] Hayawardh Vijayakumar *et al.* 2012. Integrity Walls: Finding attack surfaces from mandatory access control policies. In *ASIACCS*.

[50] Wen Zhang, You Chen, Thaddeus Cybulski, Daniel Fabbri, Carl Gunter, Patrick Lawlor, David Liebovitz, and Bradley Malin. 2014. Decide Now or Decide Later?: Quantifying the Tradeoff Between Prospective and Retrospective Access Decisions. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1182–1192.