8-2011

Beyond SELinux: the Case for Behavior-Based Policy and Trust Languages

Sergey Bratus
Dartmouth College

Michael E. Locasto
University of Calgary

Boris Otto
Dartmouth College

Rebecca Shapiro
Dartmouth College

Sean W. Smith
Dartmouth College

See next page for additional authors

Beyond SELinux: the Case for Behavior-Based Policy and Trust Languages

Sergey Bratus[d], Michael E. Locasto[c], Boris Otto[d]
Rebecca Shapiro[d], Sean W. Smith[d], Gabriel Weaver[d]

d: Dartmouth College
c: University of Calgary

Computer Science Technical Report TR2011-701
Dartmouth College

August 2011

Abstract

Despite the availability of powerful mechanisms for security policy and access control, real-world information security practitioners—both developers and security officers—still find themselves in need of something more. We believe that this is the case because available policy languages do not provide *clear* and *intelligible* ways to allow developers to communicate their knowledge and expectations of trustworthy behaviors and actual application requirements to IT administrators. We work to address this *policy engineering gap* by shifting the focus of policy language design to this communication via *behavior-based policies* and their motivating scenarios.

1 Introduction

Increasing evidence suggests that system administrators of large enterprise IT systems find themselves in an untenable position when it comes to security. For example, the National Security Agency publicly stated that it considers any large system as already compromised. Furthermore, the National Cyber Leap Year summit of 2009 recognized the inadequacy of current methods used to secure cyberspace.

Despite the availability of powerful policy mechanisms such as SELinux, AppArmor, and various hypervisor and other virtualization technologies, enterprise administrators appear to be unable to make effective use of these policy mechanisms. This indicates that there is still a significant *policy engineering gap* between the existing policy languages and the needs of enterprise administrators to express, modify, and reason about trustworthy system behavior.

The community needs a new approach to secure large, enterprise-level applications. As part of this work, we are systematically collecting data from security developers and enterprise security officers about how current state-of-the-art security policy languages fail to meet their needs. In this paper, we present some *preliminary* findings, based primarily on some initial discussions with developers.

1. We hypothesize that policy languages remain ineffective because they fail to enable *communication* of developers' knowledge of trustworthy behavior and of infosec officers's requirements.
2. We propose behavior-based policy language primitives for developers and administrators to communicate trustworthy application.
3. We discuss their potential realization as SELinux mechanism extensions.

Again, we stress these findings are preliminary; we plan to issue a more complete report when discussions and subsequent analysis are complete.

Policy Languages as a Medium for Communication We hypothesize that enterprise application security remains untenable because currently-available policy languages do not permit policy creation to easily capture developers' knowledge of trustworthy behavior.

Programs are written by developers but, in enterprise IT, must be run by administrators. The knowledge of what constitutes the intended, trustworthy behavior of a program originates with developers but must be communicated to administrators. Meaningful administration – including formulation and enforcement of security policies – is hard without such communication. For example, if the administrator has no clear idea of *what* resources or services the program legitimately needs or *when* it needs them, she cannot examine a program's resource requests and distinguish between intended and compromise-indicating behaviors.

Developers already distinguish between proper and improper behavior when they write test suites or use secure programming primitives such as privilege drop – this knowledge needs to be expressed in a useful form to administrators. However, this knowledge gets lost when an application moves from the developer to the administrator. *Policy language* primitives can and should be the vehicle of communication between the developer and the administrator (as well as the enforcement mechanism).

We also hypothesize that enterprise application security remains untenable because currently-available policy languages do not permit infosec officers to express and analyze what they regard as trustworthy-relevant behavior and properties of the systems they care about.

Behavior-Based Policy Language Primitives As a preliminary result, we propose behavior-based policy extensions that will enable developers to communicate trustworthy application *behavior* to administrators. Currently, administrators express *mandatory access control (MAC)* policies in languages such as SELinux. However, we believe describing all allowed accesses of an application via MAC may be the wrong target, and lead to policy frameworks that fail to capture the dimensions of flow and context. Indeed, we believe that context itself is particularly unappreciated. Not all access profile violations are equal; not all are equally detrimental to a policy's stated or actual security goals. The implications for security may change depending upon the sequence of accesses, intended shifts of identity in the application, or execution history of a process. Therefore, we propose shifting the focus of policy language from controlling access to communicating behavior.

Our collaboration with real-world enterprise administrators has led us to call for policy languages with the following high-level properties:

- **Clarity.** The language should enable policies that are concise, easily readable, and understandable by (human) administrators.
- **Intelligibility.** A human should be able to understand the encoded behavioral properties – and how they relate to that human's workflow – from policy itself, rather than from analysis of the rest of the system as context.
- **Machine Actionability.** The policies should be directly actionable by the policy enforcement mechanisms

This Paper In Section 2 we present a few motivating examples for behavior-based policy based upon preliminary interviews. In Section 3 we discuss the currently deployed state of the art and emergent policies. In Section 4 we describe high-level requirements for our security policy language and why we propose to use SELinux as a base language for our policy extensions and use our policy extensions in two sample policies to communicate expected process behavior and events that alter process identity. On Section 5 we discuss our ongoing work in collecting requirements from enterprise infosec officers. Section 6 orients our research

within other security policy languages and provides several historical arguments for our approach. Finally Section 7 concludes and discusses our ongoing work to evaluate these mechanisms.

2 Real-World Scenarios

Our interviews and discussions with security developers and security officers confirmed that the state-of-the art security policy mechanisms do not address the real-world security needs of enterprises. In this section, we present a few motivating usage scenarios for behavior-based policy.

Compositional Reasoning Real-world systems are often composed of several systems glued together. Interviews with information security officers revealed their need to be able to reason about a system along the dimension of this composition. Two areas in which such reasoning is important are *context dependency* and *cascading trustworthiness*.

Enterprises need to express *context dependent* security goals. A mobile sales worker may need to access the same asset with different devices via different networks. If one of these devices is a personal machine, then the company may restrict information flows that would otherwise be available had that worker connected from an enterprise device. In this scenario, there is a communication gap between the end user who needs access, the enterprise administrator, and the CISO who needs to approve the accesses granted. The trustworthy behaviors—which device can access which enterprise resources—depend upon the properties of that device and may change as new devices are introduced.

Enterprises also the need to reason about *cascading trustworthiness*. For example, when one application that is considered trustworthy invokes another application, the invoked application may need to inherit some of the authority of the invoker—and the invoker incurs trust dependence on the invokee. The communication gap between trusted application and child process means that resource privileges need to be passed from the parent process to its children. The developer needs to communicate the expected child processes of a trusted application to the IT administrator so that the administrator understands the anticipated behavior. (One approach to modeling these efforts is proposed in [9].)

Counting Primitives Interviews with information security officers also revealed their need to be able to express policy in terms of *counts*. In this way, enterprise administrators would be free to express thresholds, and in general, limit computation.

On a system level, one set of scenarios emerge when we consider the need to constrain the shape of a process subtree. The process may need to `fork()`—but not too much. In contrast, traditional formal models of security typically focus on information flow and consequently

assume monotonicity: if subject S reads object O once, it can read it arbitrarily many times. Similar problems emerge in PKI and privilege management scenarios, when trying to evaluate the semantics of varying levels of delegation and assertion, and in rootkit defense, when trying to limit the number of times some kernel table entry gets written.

Another scenario in which counting primitives are important is temporary business partner access. Enterprises need to grant temporary workers access to company resources. When an organization hires a temporary worker (end user), the enterprise admin needs to give the temp worker access to resources or services for a short period of time. This is prevalent among large pharmaceutical companies that work with high-end professionals such as physicians. However, the same doctor may be an employee or contractor with a competitor. In this scenario there is a communication gap between the temporary employee, the system administrator, and the CISO. Although some administrators may want to enumerate the behaviors to which the temporary end user should be entitled; even full-time workers cannot accurately report what they need to access to do their jobs. Rather than taking the more difficult route of tracking accesses, infosec officer might instead wish to use counting primitives to provide provide temporary workers with a *computational budget* (in the spirit of HCISEC work on "compliance budgets").

Isolation Primitives Our work with the power grid reveals the need for isolation primitives in large-scale organizations. Regional Transmission Organizations (RTOs) regulate the power operations of a variety of independent companies. Many scenarios arise here in which one party, such as an RTO, may need to establish data and control connections with another party with different standards of "trusted computer behavior"; one party might also want assurance that such connections will be limited to the relevant domain of cooperation – since these parties are also competitors [2].

3 State-of-the-art Policy Languages

Today's state-of-the-art security policy languages largely fall into two categories:

1. languages in which *developers* express policies, and
2. languages in which *administrators* express policies.

There are several different mechanisms developers can build into their software to limit potential damage caused by that software if compromised. One of the most common mechanisms comes in the form of the `setuid()` system call used by applications to drop privileges. Experimental capability-inspired systems like Capsicum [16] and UserFS [7] provide developers with a more flexible way restricting privileges of their application. These mechanisms are able to enforce behavior-based policies we have proposed; however they embed security

policy into the application's code and binary. An administrator would need to analyze the application's code to gain an understanding of the security guarantees of the software and cannot easily use the same mechanisms to enforce additional constraints.

MAC mechanisms used in practice such as SELinux and AppArmor as well as virtualization-based solutions like chroot jails, Solaris Zones, and Linux Vservers, give the administrator the ability and responsibility of correctly configuring the policies. An administrator must have knowledge of a program's resource requirements and behaviors in order to correctly configure MAC and virtualization policies. Knowledge of the specific configuration of the system is needed to properly configure these mechanisms and so these languages are only spoken by administrators – not developers.

SELinux We consider SELinux as our baseline example of the security policy languages available to enterprise administrators and describe how it is insufficient to encode the security constraints desired by real-world administrators.

We chose SELinux as our baseline for several reasons. First, SELinux is widely considered a best-of-breed policy language. Unlike many experimental policy mechanisms and systems, SELinux has found its way into the mainline OS kernel. Architecturally, it provides considerable policy strength (at least C2, with full syscall mediation [18, 5]), and is based on a solid underlying design [17] respected by both kernel developers and hackers.

We note that AppArmor (Section 6.1) is another strong baseline candidate, developed to improve on SELinux, and successfully addressing several SELinux's policy language drawbacks that emerged from user and administrator experience. However, we chose SELinux as a purer example of the architectural aspects shared by both systems.

Given a complex SELinux policy it can be extremely difficult to know what type of effect a change has on the rest of the system. This behavior runs counter to the "divide and conquer" intuition instilled in computer scientists. By that intuition, we learn to problem solve by breaking up problems into smaller pieces/modules and design APIs for these modules such that they modules can be combined to build a cohesive whole. Unfortunately, SELinux doesn't cleanly lend itself to this methodology. SELinux policies can be thought of as a big jigsaw puzzle; every pixel of the puzzle's picture must be drawn on one of the puzzle pieces. We can can think of a puzzle piece as a single domain and all the rights and privileges that come with that domain, every file and process is required to be part of some domain. If we change any puzzle piece we well end up inevitably affecting another puzzle piece.

Because of the tight interdependencies of SELinux domains, it is very difficult for humans to analyze and adapt SELinux policies as they would standard computer programs. SELinux policies often fail to translate to different systems because of differences in how the systems are laid out. Often, vendors will say that SELinux is outside the scope of their installation instructions and admins need to write a bare bones policy from scratch that may or may not reflect the actual security needs of the software. An admin needs to receive from a developer a clear, concise, understandable security policy that can be compared with a system's existing

policy in order to determine whether they are compatible and can be merged.

4 Our Approach: Behavior-Based Policy

Our scenarios motivate requirements for a behavior-based policy language.

1. Policy language should be the vehicle of communication between the developer and administrator.
2. Policy language must be able to *clearly* and *intelligibly* describe behaviors that developers consider important indicators of trustworthiness or lack thereof.
3. Policy languages should be directly actionable by the policy enforcement mechanisms.

We present several sample policies that capture common and easy to understand trustworthiness-related behaviors. For each policy we discuss why expressing it with the SELinux alone may be hard and/or produce voluminous and hard-to-understand policies.

We intend the following sample policies to be themselves usable as elements of more comprehensive real policies. For that reason, each sample policy concentrates on a single aspect of behavior. However, the point of using such an element in a larger policy is to indicate (and communicate to the policy's human readers) that behaviors that violate it *are* deemed untrustworthy.

4.1 Descendant Tree Pruning

Primitives: compositional reasoning, counting.

As noted earlier, one of the most natural characteristics of a UNIX process' behavior is the *expected* subtree of descendant processes. Developers should have no difficulty describing this tree. For most daemons, this tree is expected to be either empty or shallow. Many daemons are not expected to drop child processes at all (e.g. `syslogd`); others are only expected to `fork` off worker subprocesses or threads. Only those daemons that are expected to provide remote login shells, e.g. `sshd`, are expected to have arbitrarily complex descendant subtrees rooted at them; expectations of their behavior are not easy to capture in terms of their progeny. However, these are the exception, not the norm!

It should be noted that the typical remote exploitation scenario has long been associated with shellcodes, crafted inputs that cause the input-handling daemon to "drop shell". We note that the majority of typical shellcode attacks would violate an explicitly expressed expectation of the process' descendant subtree and therefore would be easily thwarted by a policy mechanism enforcing such expectations.

We note that the developer expectations can be conveniently expressed in a graphical form imitating the output of familiar UNIX utilities such as `pstree` or `ps -eH`. Such graphical representation can help enumerate the allowed configurations more simply than any other form such as productions, XML assertions, etc.

Furthermore, such a graphical representation can be naturally annotated with security labels attached to the input and can assist designers or administrators of distributed systems in expressing cross-system data tagging relationships.

SELinux Obstacles At first glance it appears that SELinux is well-suited for expressing the above expectations because its domain transitions are triggered by the `exec` family of system calls—so one might expect to be able to add constraints at process creation. However, clearly and intelligibly expressing expected descendant subtrees is hindered by several obstacles:

- Expressing unallowed `exec`-based transitions takes not 1 but up to 5–10 SELinux policy language statements. Moreover, at least 3 SELinux operations must be involved in the specification including the SELinux `entrypoint`, which, in our experience, has been something of a stumbling block to policy writers. While SELinux sample policies attempt to address this with providing the policy writer with a set of the M4 macro-processor macros that generate groups of related lines in concert, this trick makes adjusting or analyzing macro-expanded policies quite hard, since the origin and therefore intention of their individual lines becomes obfuscated.
- SELinux does not provide an easy way to control the use of the `fork` operation once `fork`ing has been allowed in the program's profile. Whereas it is trivial to use an SELinux policy to prohibit a process from ever `fork`ing, it is much harder to restrict the depth of `fork`ing once allowed. SELinux types do not provide a natural way of *counting* the number of uses of a permitted operation. Once an instance of an operation is permitted all future repetitions of this operation are permitted. Only switching to a different domain via an `exec` transition provides a way of limiting these repetitions in the process' progeny. As the most obvious consequence, SELinux types have no easy way of describing and thus thwarting a `fork` bomb (except for the extreme technique of forcing a process to abstain from all `fork`ing).
- An awkward kind of finite counting can be achieved with source code changes. For example, if it is desired to limit the depth of the daemon's subprocess tree to two (master process and worker subprocesses), then one can create copies of the daemon executable to be `fork`ed and provide a different SELinux label for these copies.

Thus we see that describing the shape of descendant subtrees – arguably, the most natural developer-intended behavior – is both verbose and error-prone, and is also convoluted. A better language is required.

4.2 Process Identity Change Cue Via Flag Action

Primitives: compositional reasoning, isolation

Some actions by the process signal an intended change in its requirements and expected behavior. An instance of this is the UNIX *setuid* call made to signal intended change of the *process identity*[1] [15]. For a long time, `setuid` has served "least privilege" security goals [14, 12].

During their runtime, many programs pass several distinct stages, each associated with a specific set of behaviors not meant to occur in other stages. For an example both classic and simple, consider a daemon that needs to start with a higher privilege to access needed system resources but can then drop privilege: namely, the process instance that attempts a privileged operation *after* the point of privilege drop should be deemed untrustworthy or compromised.

As another example, a common practical case of user intention that differentiates a "secure" workflow from a less "secure" one or a recreational activity is signaled by a particular easily identifiable action. Such an action is known to both the users and the policy authors as an event that demarcates security contexts: e.g., after this action, no confidential data is expected to be exchanged in a particular workflow. For example, accessing a YouTube video in a browser context suggests recreational browsing, during which 'net banking is highly unlikely and definitely not advisable; still, YouTube may well be a part of a normal workflow. Provided that in the context of an organization such semantic indications of security context switches can be easily identified, policy language should be able to express the respective changes in the expected program behavior. For example, a Flash-running browser is no longer expected to have arbitrary access to the local file system.

SELinux Obstacles Although SELinux is capable of (and indeed requires) describing programs' allowed (and/or denied) access profiles in minute detail, tying an access change to an event other than the program itself calling an `exec*` is a problem. One way of doing so includes the application itself making the discouraged `setcon()` call. Another method involves `exec`ing a specially labeled copy of the executable to effect a domain transition. Both methods are contrived and require source code modifications—we need something better.

[1] Metaphorically speaking, the caterpillar, the pupa, and the butterfly may be the same organism working under the same program, but caterpillars are not expected to fly, butterflies are not expected to eat through leaves, and pupae are not expected to move; if they do, something has probably gone very wrong with the program.

5 Business Community Interviews

The paper addresses the research question whether state-of-the-art security policy languages meet the needs of the business community. As noted earlier, in ongoing work, we are exploring this question, using a qualitative, empirical research design [3, 10]. We are conducting expert interviews with Chief Information Security Officers (six so far) based on a structured interview guideline. Expert interviews are considered an appropriate method in particular in early exploratory phases of a research effort because they allow for shortening time consuming data gathering when the experts function as "crystallization" points for practitioner knowledge [11, 13].

The interview guideline[2] covered five topical clusters:

- Strategic and environmental context
- Organizing information security management
- Information systems perspective
- Information technology and sources of trust evidence
- Trust evidence scenarios

The clusters were further divided into topics. In total the interview guideline comprises 14 topics.

Table 1 shows the expert interviews used for data collection.

All interviews were documented by the interviewer (with the exception of the representative of BT who filled in the guideline on his own) and sent to the interviewee for clarification of open questions.

Data analysis followed the principles of qualitative text analysis and was inspired by the constant comparison method introduced by Glaser [6]. From the data analysis emerged policy engineering scenarios and overall requirements for trust languages.

The sample size, of course, does not support statistical significance—which is not the goal of the study. In contrast, the paper aims at exploring fundamental concepts in the business community which should then be elaborated and tested in subsequent research efforts.

[2] `http://www.tuck.dartmouth.edu/digital/research/project-detail/intel-trust-project/`

Participant organization	Date and time	Location	Industry	Country	Revenue 2010	Expert role
1	03/08/11: 2-3 pm	on-site	Higher Education	US	n/a	CISO
2	03/30/11: 8-9 am	telephone	Pharmaceutical	US	68 bn. USD	Manager Risk Management and Quality Assurance
3	03/21/11: 2-3 pm (filled in questionnaire sent on 04/20/11)	telephone, email	Telecommunications	UK	21 bn. GBP (2009)	Program Security & Compliance Director
4	04/21/11: 9-10 am 04/28/11: 10-11 am 05/06/11: 10-11 am	telephone	Electronics and electrical engineering	DE	76 bn. EUR	Corporate Information Security Manager
5	06/16/11: 10-11.30 am	telephone	Medical technology	CH	9 bn. CHF	CISO
6	07/20/11: 10.30-12 am	telephone	Insurance	CH	29 bn. USD	Head Information Security Awareness and Reporting

Table 1: Expert interview details.

6 Related Work

In this section we discuss some related security policy languages and mechanisms. We then provide historical arguments for our behavior-based policy approach.

6.1 Related Security Policy Languages

The policy languages of state-of-the-art kinds of security policy mechanisms—most notably SELinux and AppArmor, but also experimental capability-based systems, virtualization-based solutions like Solaris Zones or Linux Vservers, and even the latest hybrid sandbox-like systems like Capsicum—do not make it easy to express simple enough statements regarding the program's behavior.

We note that the designers of AppArmor saw significantly simplifying the language of application profile definitions in comparison to that of SELinux policies as a major step towards improving both the usability and the practical security of the overall system (e.g., see [8] for a discussion). AppArmor also identified several common scenarios that could not be naturally expressed with SELinux policies, and added features to describe them (e.g., confinement at sub-process granularity). Still, we believe that there are conceptually simple but trustworhiness-relevant behaviors that are hard to express even after with AppArmor's

simplifications and added features.

`Setuid` as a policy tool. The problem of process identity change is old, and the `setuid` mechanism of signaling the change from *inside* the process, in the program's code is perhaps the most historically effective means or preventing Internet daemon compromises.

The initiation of this signal event from inside the modern UNIX's rich process contexts turns out to be non-trivial to both implement correctly and use correctly; subtle pitfalls abound (e.g., Chen et al. "Setuid demystified" [4], and the follow-up [15]).

6.2 Historical Arguments for Behavior-Based Policy

Firewall Policies and HIPS We notice a common trend in firewall policies and systems policies, which we attribute to pressure to accommodate descriptions of behavior into policies: modern firewalls dynamically modify their sets of rules, adding and removing them based on certain network events they observe. This trend persists across vendors and technologies, and can be demonstrated on such diverse systems and mechanisms as Cisco IOS reactive ACLs, Linux Netfilter connection tracking modules, and others.

Firewall policies, we believe, lead in this trend, due to starting with much less descriptive power and available context information for their decisions, which necessitated the introduction of *dynamic* elements earlier than in HIPS, which enjoyed richer OS-informed contexts and higher available computational power.

OS policies such as SELinux remain largely static and *"compiled"* into low-level structures such as SELinux's access vectors, designed for minimal interpretation at the point of application (e.g., reduced to LSM syscall hook callback access vector checks).

From "compiled" to "interpreted" policies. We believe that the future, however, belongs to "interpreted" policy logic that is run concurrently with the process in a trusted process (which acts as a reference monitor), and relies on much richer data structures contained in a "policy computational environment". This approach should provide much better sensitivity to needed updates, necessary policy changes, etc., without a recompilation (and a costly reboot).

Indeed, the need for modifying policies "on the fly" to accommodate the changes in process identity and mutability is getting realized by practitioners, and necessitates moving policy decision logic into trusted processes, hypervisors, etc., where the policy is *"interpreted"* and dynamically changed (e.g., [1, 2]), with SELinux or LSM hooks used simply to trap syscalls of interest and to protect the trusted dynamic policy interpreter processes.

6.3 Trust Distribution Diagrams

One promising approach to easing the job of a policy engineer is to give her the ability to cleanly express trust relationships. One obstacle to doing so is the lack of design tools that can help developers visualize and structure trust relationships between different components and different software artifacts. Graphical policy design tools can help overcome this obstacle; graphical approaches to software design (i.e., UML, data flow diagrams) are already popular. One of the central useful concepts of such a graphical approach to trust design is to show how trust relationships might change in response to input or under new conditions.

Trust Distribution Diagrams (TDDs) [9] seek to address these types of problems by making the trustworthiness properties of a system more explicit (and thus amenable to structured analysis). The relationship of trusted components, their communications paths, and data dependencies moderates *design*–level risk. The goal of TDDs is to define a graphical language for expressing the distribution, amount, and migration of trust in designlevel components and understand its application to the craft of security architecture.

TDDs help express a developer's assumptions about trustworthiness properties and trust relationships between system components, including how they evolve over time in response to input or other events.

In particular, TDDs can generalize both the process identity-related policies discussed in 4.2 and extend the process tree-related primitives of 4.1 from parent-child process relationships to richer communication and trust relationships. They can enable developers to state the *consequences* for violating expected behavior by selected components, and thus control the system's failure modes.

7 Conclusion

Our ongoing fieldwork with security developers and enterprises has made us aware of the need for a behavior-based policy language that is clear, intelligible, and machine-actionable. Our preliminary results suggest three policy primitives: compositional reasoning, counting primitives, and isolation primitives. We then proposed to realize these isolation primitives as SELinux extensions and provided two simple policies to communicate expected process behavior and trust-altering process events. We believe that the future security policies, as evidenced by firewall policies and the increased popularity of interpreted languages, will facilitate communication among developers and administrators via a stateful, interpreted language.

We are currently refining our proposed extensions into more formal policy languages and refining our practitioner interview data into more precise scenarios; we then plan experiments to quantitatively and qualitatively evaluate the relative effectiveness of these current and proposed policy languages for communication of desired properties of trustworthy system

behavior. We are also exploring the problem of *attestation* languages, as a dual to policy languages.

We reiterate that this paper is preliminary; we plan subsequent reports as the work progresses.

Acknowledgments

We would like to thank our security industry contacts who helped connect us with their clients' information officers and provided a unique perspective on the current and emerging needs and challenges of enterprise IT security. For confidentiality reasons, we do not reveal their identities in this TR, but express profound gratitude for their significant contributions.

This research was supported in part by Intel Corporation, which does not necessarily agree with the views and conclusions presented.

References

[1] Berthold Agreiter, Masoom Alam, Michael Hafner, Jean-Pierre Seifert, and Xinwen Zhang. Model Driven Configuration of Secure Operating Systems for Mobile Applications in Healthcare. In *ACM Workshop on Model-Based Trustworthy Health Information Systems*, 2007.

[2] Katelin A. Bailey and Sean W. Smith. Trusted Virtual Containers on Demand. In *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing*, pages 63–72, 2010.

[3] R. Y Cavana. *Applied Business Research: Qualitative and Quantitative Methods*. John Wiley & Sons Australia, Milton, 2001.

[4] Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, 2002.

[5] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 225–234, 2002.

[6] B. G. Glaser. The Constant Comparative Method of Qualitative Analysis. *Social Problems*, 12(4):436–445, 1965.

[7] Taesoo Kim and Nickolai Zeldovich. Making Linux Protection Mechanisms Egalitarian with UserFS. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[8] Achim Leitner. Novell and Red Hat Security Experts Face Off on AppArmor and SELinux. *Linux Magazine*, (69):40–42, 2006.

[9] Michael E. Locasto, Steven J. Greenwald, and Sergey Bratus. Trust Distribution Diagrams: Theory and Applications. In *Proceedings of the 4th Layered Assurance Workshop (LAW 2010)*, Austin, TX, December 2010.

[10] J. A. Maxwell. *Qualitative Research Design: An Interactive Approach.* SAGE Publications, Thousands Oaks, 1996.

[11] M. Meuser and U. Nagel. The Expert Interview and Changes in Knowledge Production. In A. Bogner, B. Littig, and W. Menz, editors, *Interviewing Experts*, pages 17–42, Hampshire, UK, 2009. Palgrave Macmillan.

[12] Robert A. Napier. Secure Automation: Achieving Least Privilege with SSH, Sudo, and Suid. In *LISA '04: 18th Large Installation System Administration Conference*, pages 203–212, 2004.

[13] J. Reitman Olson and K.J. Biolsi. Techniques for Representing Expert Knowledge. In K. A. Ericsson and J. Smith, editors, *Toward a General Theory of Expertise*, pages 240–285, Cambridge, UK, 1991. Cambridge University Press.

[14] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[15] Dan Tsafrir, Dilma Da Silva, and David Wagner. The Murky Issue of Changing Process Identity: Revising "Setuid Demystified". *USENIX ;login:*, 3:55–66, 2008.

[16] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[17] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, 2002.

[18] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, 2002.