Assurance for Defense in Depth via Retrofitting

Vinod Ganapathy
Rutgers University
vinodg@cs.rutgers.edu

Trent Jaeger
The Pennsylvania State University
tjaeger@cse.psu.edu

Christian Skalka
University of Vermont
skalka@cems.uvm.edu

Gang Tan
Lehigh University
gtan@cse.lehigh.edu

ABSTRACT

The computer security community has long advocated *defense in depth*, the concept of building multiple layers of defense to protect a system. Unfortunately, it has been difficult to realize this vision in practice, and software often ships with inadequate defenses, typically developed in an ad hoc fashion. Currently, programmers reason about security manually and lack tools to validate assurance that security controls provide satisfactory defenses. In this position paper, we propose STRATA—a holistic framework for defense in depth. We examine application of STRATA in the context of adding security controls to legacy code for authorization, containment, and auditing. The STRATA framework aims to support a combination of: (1) interactive techniques to develop *retrofitting policies* that describe the connection between program constructs and security policy and (2) automated techniques to produce optimal security controls that satisfy retrofitting policies. We show that by reasoning about defense in depth a variety of advantages can be obtained, including optimization, continuous improvement, and assurance across multiple security controls.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Experimentation, Management, Security

Keywords

Automated Retrofitting, Defense in Depth, Authorization, Containment, Auditing, Assurance

1. INTRODUCTION

The security community has long encouraged programmers to strive to implement *defense in depth*, where multiple layers of security controls are employed to protect security-sensitive operations. However, programmers almost always focus on functional and economic issues initially, often delaying the introduction of needed security controls until after initial deployment. As a result, programmers often find themselves in the so-called penetrate-and-patch mode, removing vulnerabilities as they are identified by adversaries.

Even when programmers decide to add the security controls necessary to implement defense in depth, they face many practical challenges. First, retrofitting software with a security control requires a comprehensive understanding of the program code and security requirements to integrate the security controls correctly. For this reason, past projects that retrofit software manually for attack containment [53, 28, 55], authorization [74, 70, 19, 4, 68, 50, 32, 11, 56], or auditing [5, 3] often span multiple years. Gaining consensus over whether the implementation of a security control is satisfactory is ad hoc [59, 75, 58], and mistakes have been made [20, 76], some of which were not discovered until years later [65]. Second, retrofitting software with security controls must account for several other factors beyond security. Naive containment implementations can cause tremendous performance overheads and naive authorization mechanisms can result in spurious policy management. As a result, many attack containment projects, particularly for performance-critical software such as operating systems, have been abandoned and some authorization mechanisms have been refined several times after introduction [49]. Third, security controls may interact, making other controls unnecessary or impacting their placement. For example, if authorization prevents data leakage provably (noninterference [26]), then there is no need to audit for data leaks. However, programmers lack tools to reason about such interactions, resulting in unnecessary or misplaced security controls.

Researchers have recognized that programmers need assistance in retrofitting their programs with security controls, but the proposed automated methods still require too much programmer effort, fail to account for factors other than security, and do not reason about multiple security controls. First, researchers have proposed automated methods to detect missing or misplaced authorization controls [76, 20, 65, 61, 63, 51, 64] or even retrofit programs for authorization [25, 45, 38, 62], but these methods require detailed program knowledge, such as security-sensitive data types, variables, and/or statements, or require a partial authorization implementation. Second, methods to retrofit for authorization and attack containment [36, 10] have been naive about the impact of security controls on other factors, such as the performance and management overheads, preventing wide adoption. Third, we are not aware of any prior work that reasons about retrofitting for auditing or for multiple security controls and their interactions.

Our insight is that advances in retrofitting software for security enable the development of a holistic framework for the assurance of security controls for defense in depth. While retrofitting programs for security is a challenging problem for any security control, recent advances in methods for retrofitting programs for security demonstrate what can be automated and how, distinguishing what can be computed from what intelligence programmers need to supply. The proposed approach takes a comprehensive view of the problem, with an emphasis on *automated* and *interactive* tools that developers can use to identify site-level security goals, explore the design space of security mechanisms, and retrofit legacy code to enforce security policies in a manner that can be machine-verified for assurance.

In this position paper, we examine the unification of three common security mechanisms — containment, authorization, and auditing — to assess how reasoning about defense in depth encompassing these three mechanisms may improve security assurance. First, we find that placing security controls for these mechanisms involves solving a set of common problems, so designing methods to solve those problems and to verify the effectiveness of the solutions may be reused. Second, we find that we can compose the validation of defense in depth for this combination of security controls, enabling assurance for defense in depth. Third, we find that runtime auditing can be leveraged for continuous improvement of the placement of security controls for defense in depth, ensuring that the controls can be optimized for the desired policies. We refer to completed research results where available, but a goal of this paper is to motivate reuse of common ideas across controls and integration of controls for improved security.

The remainder of the paper is structured as follows. In Section 2, we examine the problem of designing programs to control access to program and system resources using multiple security controls. In Section 3, we provide an overview of how to use automated retrofitting of programs to produce validated security controls for defense in depth. In Sections 4 to 6, we explore the challenges in retrofitting programs for containment, authorization, and auditing independently. In Section 7, we outline the problem of unifying retrofitting methods for defense in depth and examine opportunities for assurance of defense in depth, including continuous improvement. In Section 8, we conclude the paper.

2. BACKGROUND

2.1 What Should Retrofitting for Defense in Depth Do?

When program vulnerabilities become too numerous, programmers may be motivated to make fundamental changes to their programs to add security controls. For Sendmail and OpenSSH, programmers found that the typical penetrate-and-patch approach to security was not keeping them ahead of adversaries, leading to complex retrofitting [53] or complete reimplementations [52, 6]. For programs that process resources belonging to multiple clients, such as servers and middleware, programmers often found that simple isolation approaches (e.g., sandboxes) were insufficient to protect data security and provide necessary functionality [15, 21]. We use the simple program below to demonstrate the problems.

```
request_loop (client_data, private_data) {
  read(client_passwd, client_req );
  if (necessary ||
      compare_client(client_passwd,
                     private_data))
    access_object(client_req, client_data);
}
```

The client request loop above is representative of many programs that require retrofitting. This program processes requests from multiple, mutually-untrusting clients (obtained by `read`) by: (1) comparing a client-supplied password (`client_passwd`) to the program's password database (`private_data`) in `compare_client` and (2) processing a client request (`client_req`) to access data managed by the program (`client_data`) in `access_object`. In this discussion, we assume that the program code is benign, but may have flaws that allow client input read by the program to permit unauthorized access. The first operation may cause vulnerabilities if the program allows client input to affect the program's passwords or if some password data is leaked as a result of the comparison. The second operation may cause vulnerabilities if it allows any client unauthorized access to the client data of another client. Many programs perform these two types of operations, including operating systems, middleware, server programs, and even some user applications. For example, operating systems process many client requests (e.g., system calls) and process private operating system data that must not be manipulated by clients. On the other hand, browser applications also run programs from multiple sources (i.e., the browser's clients), so they must control access to browser resources available to those programs and protect their private resources from leakage and unauthorized modification.

In this discussion, we will focus on retrofitting programs to control client access to *security-sensitive operations*, such as those in the program above that use the program's private data and client data.

We examine three kinds of security controls that are commonly used to achieve this goal. First, programmers may use *containment* to place protection boundaries that limit the ways that clients may access security-sensitive data. For example, the program above may be privilege-separated [57] into two modules running in separate processes: (1) one that receives client requests and provides access to client data using `access_object` and (2) another that runs `compare_client` that has access to the private data. Clients can only communicate directly with the first module, limiting the program flows that may reach or leak the private data.

Second, programmers use *authorization* to control access to program data. For example, the program above may be retrofitted with a reference validation mechanism that satisfies the reference monitor concept [7] to ensure correct enforcement of an access control policy governing which clients may access which client data and preventing leakage and unauthorized modification of private data, regardless of the complexity of the code in the `compare_client` and `access_object` functions. Reference validation mechanisms must be designed to enforce the data access policies expected by the programmer, whose goals may include least privilege [57], lattice policies [16], noninterference [26].

Third, programmers use *auditing* to collect information to aid intrusion detection retroactively for authorized operations. For example, clients authorized to run `compare_client` may still cause the private data to be leaked through some program flaw, so auditing could record the values of the authorized operation and the data returned to the client to enable later detection of whether leakage occurred. As can be seen, these security controls form three layers of defense, where containment limits client access at the boundaries, authorization within the program, and auditing follows authorized operations.

2.2 State-of-the-Art in Retrofitting Programs for Defense in Depth

Programmers retrofit programs with containment [53, 28, 55], authorization [74, 70, 4, 68, 19], and auditing controls [5, 3] manually, which presents a variety of challenges. First, programmers must identify security-sensitive operations from low-level program constructs, such as variables, data types, and statements. While the program above may be simple, real programs have hundreds of user-defined types and thousands of program statements. Despite the availability of several prototype reference monitor implementations, the Linux Security Modules framework [74] still contained several errors [20, 76], even some that were not discovered until years later [65]. Second, programmers must determine where to apply controls to protect those security-sensitive operations, but they must be careful not to introduce high performance and management overheads. Programmers currently balance such functional and security requirements in an ad hoc way. If the variable `necessary` in the `request_loop` is usually true, then separating `compare_client` may be satisfactory, but otherwise large overheads may be incurred. As a result, retrofitting projects take several years, face delays that bring their purpose into question [59], and may reduce the scope of security controls to only known attacks [53]. Because of these challenges, only a few programs have been retrofit with all three security controls we investigate in this project.

Researchers have recognized the challenges facing programmers, but to date fail to address the most fundamental of those challenges. First, proposed methods to retrofit code still require programmers to identify security-sensitive operations from low-level code constructs, such as code patterns, data types, and variables that correspond to such operations [25, 61, 38, 10]. For example, to automate placement for information flow control, all the relevant variables must be identified and assigned an accurate security label. Second, most current research only addresses functional concerns implicitly if at all. For example, some prior work aims to produce a "minimal" number of security controls [38, 48], but these methods still introduce far more controls than added manually. Recent research has proposed a retrofitting method that uses functional and security constraints as input [29, 30]. However, such constraints are written as traces in terms of program locations, still requiring significant program knowledge to get right (e.g., context sensitivity). Finally, none of the prior methods retrofit software with multiple security controls, possibly introducing spurious security code.

2.3 Goals for Retrofitting Programs for Defense in Depth

The goal is to develop a method that programmers can use to retrofit their programs with security controls for containment, authorization, and auditing that satisfy explicit security goals (e.g., policies to enforce) and are globally optimal relative to functional costs. Thus, we have two broad challenges: (1) develop the theory and techniques to retrofit multiple security controls optimally from program code, security goals, and functional concerns and their costs; and (2) reduce programmers effort for producing security goals and the costs of functional concerns sufficient to achieve desirable security in practice. Based on the limitations of prior research, we highlight the essential questions presented by these challenges:

- Can we design methods for identifying security-sensitive operations and security goals for programs that do not require detailed, manual analysis of program code?
- Can we design methods to retrofit programs with containment, authorization, and auditing security controls that enable verification that each type of security control enforces a program's security goal with respect to that program's security-sensitive operations?
- Can we design methods to optimize the functional cost of satisfying a security goal across containment, authorization, and auditing simultaneously?

Thus, an ideal retrofitting method would extract security-sensitive operations from `request_loop` and relate those operations to security goals, with detailed, manual code inspection or annotation by programmers. Using the security-sensitive operations and security goals, this ideal method should produce a retrofitted program that consumes minimal functional cost and verifiably satisfies those security goals for the security-sensitive operations.

3. APPROACH OVERVIEW

The goal is to retrofit a program to add a series of defensive security controls to protect program data from unauthorized access, specifically containment, authorization, and auditing. While there are many differences among these security controls, we find that retrofitting these security controls into programs requires solving four common problems:

- *Finding security-sensitive operations.* Each security control aims to mediate access to security-sensitive operations for different purposes (e.g., defining protection boundaries or logging such operations). While security-sensitive operations may differ for individual controls, we find that such operations share the ability to direct execution among unsafe choices. We propose a method based on finding the program statements where control and data "choices" are made using input from untrusted sources [48].
- *Relating security-sensitive operations to security goals using retrofitting policy.* We have found that simply mediating every security-sensitive operation creates unnecessary overhead for performance and policy management. Instead, programmers need a way to relate security goals to security-sensitive operations that does not require detailed, manual analysis of program code. We propose a method that programmers use interactively to find relationships between security-sensitive operations based on their impact on satisfying security goals [49], which we call a *retrofitting policy*.
- *Place controls for security goals.* Given a retrofitting policy, the goal is to transform the program to satisfy that policy while minimizing cost. While different transformations are applied for different types of security controls, choosing where to place security controls requires complete mediation of relevant program flows in all cases. We propose to explore use of *program dependence graphs* [23] (PDGs) for reasoning about control and data flows uniformly for all security controls.
- *Verifying correct transformations.* Despite the use of different methods for placing transformation and distinct transformation primitives, each method transforms code to mediate security-sensitive operations for a security goal. We explore how to leverage formal methods, so that high assurance is obtained from our retrofitting framework. We plan to verify the correctness of the transformation methods proposed by building proofs of correctness inside Coq [13].

Figure 1 presents an overview of our proposed STRATA framework, which aims to implement the methods described above. The STRATA framework enables programmers to retrofit their programs with containment, authorization, and auditing security controls in two steps. In the first step, programmers interactively develop retrofitting policies for each of these security controls, leveraging methods to find security-sensitive operations and relate those operations to security goals for retrofitting policy. In the second step, automated methods transform programs with security controls to

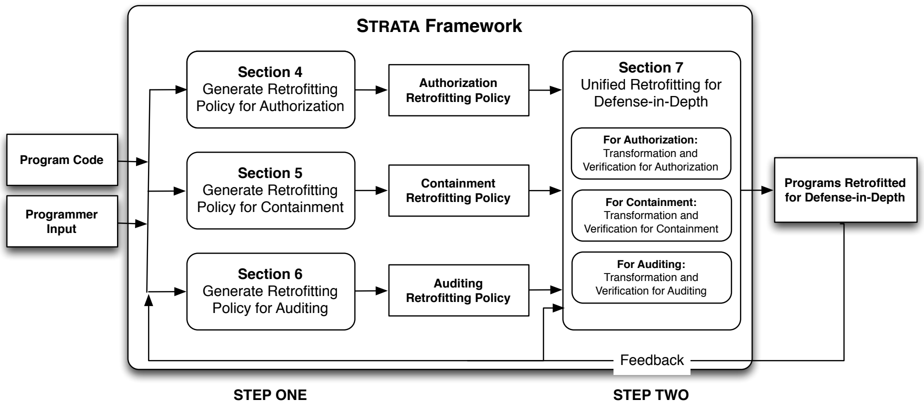
STRATA Framework
Section 4
Generate Retrofitting Policy for Authorization
Section 5
Generate Retrofitting Policy for Containment
Section 6
Generate Retrofitting Policy for Auditing
Authorization Retrofitting Policy
Containment Retrofitting Policy
Auditing Retrofitting Policy
Section 7
Unified Retrofitting for Defense-in-Depth
For Authorization: Transformation and Verification for Authorization
For Containment: Transformation and Verification for Containment
For Auditing: Transformation and Verification for Auditing
Program Code
Programmer Input
Programs Retrofitted for Defense-in-Depth
Feedback
STEP ONE
STEP TWO

Figure 1: STRATA framework for assurance of security controls for defense in depth

satisfy a composition of those retrofitting policies while minimizing cost and verify the correctness of such retrofitting across all three security controls.

4. RETROFIT FOR AUTHORIZATION

Historically, there are two kinds of related, but distinct, security models for authorizing operations on data. First, *access control* mediates access to a program operation based on who requests the access (subject) and what the data is (object), as well as the operation itself. In the `request_loop`, access control limits which clients can perform which requests (operations) on which objects in `access_object`. Access-control policies can be represented by access-control lists or capabilities of access-control matrices [43], which are intuitive to understand and easy to enforce. As a result, they are widely adopted in programs such as servers and operating systems. The reference monitor concept [7] defines the requirements for enforcing an access control policy correctly: complete mediation, tamperproofing, and verifiability. For example, complete mediation requires the placement of *authorization hooks* in programs that mediate all security-sensitive operations [74, 70, 19, 4, 68, 32, 56, 50]. However, even complete mediation has been difficult to ensure because programmers do not identify security-sensitive operations explicitly.

Second, *information flow* tracks the propagation and release of data through the program. Rather than being focused on program operations, information flow is concerned with how data associated with particular security properties may influence other data values, causing security-critical data to be modified by untrusted data or leaked to untrusted subjects. In the `request_loop`, information flow control limits how private data (e.g., password database) may be used to prevent its leakage when comparing it to client input (e.g., input password) in `compare_client`. These influences are either through explicit data flows or implicit flows [16], in which data is leaked via control structures of programs. For security, hooks must mediate all unauthorized information flows using declassifiers (for secrecy) and endorsers (for integrity) that remove offending data from those flows.

We aim to retrofit software for both access and information flow control. Manual placement of authorization hooks and mediation for information flow control is laborious [31, 37, 59] and error-prone [20, 76, 65], so we propose to develop semi-automated methods for this purpose.

The substantial limitations of previous systems are that they offer little help to programmers for identifying security-sensitive operations or the relationships between security-sensitive operations and security requirements, which we call a *retrofitting policy*; previous systems often assume the retrofitting policy to be a manual input [29, 30, 45]. The reality, however, is that very often programmers have only a rough idea about the security implications of individual program statements and are unwilling to spend time to identify such information manually. We believe that techniques that help programmers discover retrofitting policies are needed.

Approach Overview. We define a retrofitting policy to be a set of *connections* relating basic *constructs*. For access control, basic constructs are the program statements that correspond to security-sensitive operations; an example connection is the operation subsumption between two operations, meaning that authorization of the first operation always implies the authorization of the second. In an information-flow policy, the basic constructs are also security-sensitive operations, which are those program statements that operate on sensitive information at data sinks. Subsumption is also an example of a connection between sinks meaning that the declassification (sanitization) of the first sink always implies declassification (sanitization) of the second.

In our recent work [48], we proposed a technique that infers security-sensitive operations for access control from the least amount of programmer input of any known method. A similar method has since been proposed for Android analysis [46]. Our method requires only the identification of language-specific lookup functions and the sources of untrusted input. Its inference technique is based on a key observation: security-sensitive operations correspond to the deliberate choices the program makes using client input for retrieving data from data collections and for selecting the conditional code paths for processing that data. Therefore, the technique tracks the "choices" made by client requests to automatically infer security-sensitive data and operations. Experiments performed on programs such as `X server` and `postgres` demonstrate this technique is effective at identifying almost all security-sensitive operations. We compared our results with manual hook placements by experts.

However, identifying security-sensitive operations alone is not enough for efficient hook placements. Human experts often remove unnecessary hooks using domain knowledge. As a result, the only way to reduce the number of hooks is to make the domain knowledge explicit. In an ongoing work [49], we make the domain knowledge explicit through the definition of retrofitting policies as connections between security-sensitive operations. We have identified two kinds of connections: operation subsumption and operation equivalence. We have discussed operation subsumption before. Operation equivalence means that two operations are always authorized for the same set of subjects. For multilevel security [9] (MLS), two operations that read the same object are equivalent because the same set of subjects will also be authorized. These connections are utilized for removing unnecessary hook placements. For instance, if operation one subsumes operation two and if operation one dominates operation two in control flow, then the authorization hook for operation two is unnecessary (assuming an authorization hook for operation one is already there). Preliminary experiments on a variety of software, including the X server, databases, and the Linux kernel, demonstrate our methodology can reduce programmers' effort for discovering their intended policies and can already reduce the number of access control hooks by 30% [49].

To further improve the reduction, more automatic methods are needed to discover retrofitting policies. To help programmers find relevant connections, STRATA must provide security and performance constraints and analyses that find pairs of security-sensitive operations that satisfy those constraints to suggest connections automatically (e.g., must enforce MLS policies, described above). If programmers agree with the high-level constraint, then they can select the resultant connections as a group, rather than one at a time. To date, we have proposed two constraints, one for security and one for performance. Given this experience, we feel it is necessary to investigate the effectiveness of other constraints and other methods for using those constraints. For example, roles [22] define groups of permissions that are authorized for the same subjects, enabling the computation of equivalence relations. Sun *et al.* require role specifications as input [64], but probably role mining [24, 47] will be useful for this problem.

We also feel that this approach applies to the setting of information flow. However, mediators for information flow perform different tasks than access control hooks, so the semantics of their connections will differ. For example, declassifiers allow subjects to receive a subset of the flow's data. Further study is needed to investigate techniques to suggest possible types of declassifiers to programmers, based on types of sources and sinks and types of data that flow between them. One example suggestion is to say that flows between source s1 and sink d1 and between source s2 and sink d1 should use the same declassifier. Given these suggestions, programmers then decide what declassifier to apply and provide the actual declassification code (for example, the code for sanitizing SQL strings).

In addition, Strata needs multiple program analysis and transformation techniques to place authorization hooks automatically in programs. Given a retrofitting policy, the next step is to compute a minimal cost placement for authorization code that satisfies that policy. Previous work on access control hook placement relies on control dependence, whereas previous work on mediator placement relies on data dependence. However, information flow control often leads to fine-grained, non-intuitive mediation requirements, particularly for implicit flows [37]. In general, we believe considering both control and data dependence will be synergistic, but how exactly they interact for better hook placement will be a research question (especially in the case of implicit flows).

We feel that a uniform program representation, called program dependence graphs (PDGs [23]), will help this problem. A program's PDG represents both the control dependence and data dependence of the program and can be extracted using efficient program analysis. We believe the benefit of PDGs is that it will enable a unified framework to compute better hook placements for a variety of security goals, from access control to explicit information flows to implicit information flows. For access control, the framework uses mostly the control dependence to reduce the number of hooks, but data dependence can be used to satisfy complete mediation without blocking authorized operations. For controlling explicit flows, the framework considers data-dependence edges to insert code that tracks data flows and taint checks or declassifiers at appropriate places. Implicit flows can be taken into account by considering both control and data dependence [38].

5. RETROFIT FOR CONTAINMENT

Experience shows that despite our best efforts at improving software security, adversaries may bypass defenses to achieve malicious goals. This is because software is often retrofit for security manually, using ad hoc techniques. These techniques are error-prone, and may leave avenues that adversaries can later exploit [20, 76, 65]. Vulnerabilities may remain despite our best efforts to retrofit software for security, especially if the requirements used during the retrofitting process evolve over time [33, 42]. Robust software assurance must therefore include a layer of defenses that confine adversaries, even if the system is compromised.

Much of today's software is not written with the goal of confining adversaries. Most server applications as well as systems software are written as monolithic artifacts. Vulnerabilities in such systems have been exploited by adversaries to gain control over the entire server [53]. Until about five years ago, web browsers were also designed as monolithic systems. In such designs, the browser kernel, script parsers, renderers, and third-party plugins ran within the same protection domain. This design lead to many security attacks, wherein a vulnerability in a plugin often gave an adversary complete control over the browser [27, 69, 66, 54]. Motivated by such attacks, the browser industry has shifted its focus to *compartmentalized* or *modular* designs, in which browser subsystems execute within different protection domains. For example, Google Chrome uses different OS processes to sandbox web content on each tab, and also creates new OS processes to execute plugins and other third-party browser content.

Strata also follows this approach, and aims to automatically modularize software to enable attacker containment. Our overall goal is to retrofit a monolithic software system to adhere to two basic security principles: (1) *Privilege Separation*, which posits that resources that require different access rights must execute within different protection domains, and (2) *Least Privilege*, which posits that each module, running within its own protection domain, must only receive the privileges that it needs to accomplish its task. Together, these two principles ensure that the attack surface of the modularized software system is minimized, limiting the damage that an adversary can inflict if he were to obtain access to the system.

Strata will develop a number of techniques to retrofit software for attacker containment. First, it will provide a rich interface to specify resources that must be protected. Each of the specified resources will be contained within their own protection domain. Second, Strata will investigate efficient techniques to modularize software. The main performance cost of modularization is that method invocation, which is inexpensive in a monolithic software system, involves crossing protection domains. Strata rectifies this by optimizing for performance within the restrictions of the retrofitting policies. Third, in contrast to prior techniques that primarily used OS processes to define protection domains, Strata will consider a number of alternatives, including language-based protection, lightweight virtual machines, as well as enhanced OS APIs as protection domains, and will develop transformation techniques tailored to these domains. Strata will also include verification techniques that provide assurance on the correctness of the modularized code.

Approach Overview. In a retrofitting policy for software modularization, we identify connections between security-sensitive operations (as done for authorization). However, in this case the concept of a connection has a negative connotation—A connection between two security-sensitive resources requires the two operations to be isolated in individual protection domains. Using the web browser as an example, a developer may identify a connection between the browsing history and network operations because browsing history should not be leaked over the network. The developer can then use the control and data dependencies from the PDG to iteratively identify statements in the software system that are connected and must therefore appear in separate protection domains.

In Strata, a developer additionally provides a performance cost model as part of the retrofitting policy. This model will allow developers to identify code paths that are frequently executed, e.g., using information gathered from runtime profiles, as well as the cost of crossing protection domains. The performance cost model identifies resources that could potentially be placed in the same protection domain to provide good performance in the modularized software system. The connections and cost model together provide a candidate retrofitting policy that is refined iteratively using code analysis. Such performance cost model can potentially be obtained automatically from profile information gathered at runtime.

Strata uses the candidate retrofitting policy identified above, together with static code analysis to identify possible module boundaries in the monolithic software system. This analysis has two goals. First, the modules identified by the analysis must satisfy all the connections, i.e., all program constructs manipulating connected resources *must* be isolated in separate modules. A program construct may be involved in accessing a pair of connected resources; in such cases, it may have to be replicated, with each replica serving one resource. Second, the overall performance cost of the modularization should be small. Program constructs manipulating unconnected resources can be co-located in the same protection domain, provided that the resulting costs of domain crossings is not excessive.

Strata casts the problem of identifying module boundaries as an optimization problem on the inter-procedural control-flow graph (CFG) of the monolithic software system. The edges of the CFG are annotated with weights from the performance cost model. The goal of the optimization then is to partition the graph into subgraphs so that: (a) nodes labeled with connected resources appear in separate subgraphs, and (b) the sum total of the edge weights crossing subgraphs is minimized. Nodes labeled with unconnected resources can appear in the same subgraph, and because there may be several such pairs of unconnected resources, the analysis has some flexibility in identifying module boundaries. Strata formulates this problem as one of finding a min-cost multicut in the directed graph denoted by the CFG [38].

Strata presents a ranked list of such boundaries, together with the estimated cost of the associated modularization, to the developers. The developers may interactively explore this design space before settling upon a retrofitting policy. Once the retrofitting policy has been identified, the developers must also supply a *security policy*, which specifies the permitted message interactions between modules separating a pair of connected resources. Such a policy could be developed interactively. The developers provide a candidate security policy, which will imply a set of security-sensitive operations and connections, resulting in a candidate retrofitting policy. The developer then iteratively refines the policy by observing the runtime behavior of the retrofitted system.

Once the developer has identified module boundaries, Strata transforms the code to enforce those boundaries. The main challenge in this step is to map software artifacts to the specific isolation primitives used, and to generate code to enable communication across protection domains. For example, if OS processes are used to modularize the system, Strata have to generate marshaling and de-marshaling code in each module, together with calls to the OS's IPC primitives to enable communication. Strata will include a variety of isolation primitives, including OS processes (as in PrivTrans), Capsicum sandboxes [71], lightweight VMs and transactions [17], and language-level primitives, such as JavaScript's Harmony modules [1]. Strata's transformation component also generates code to enforce the security policy specified by the developer at module boundaries.

In preliminary work, we have developed a prototype of the above approach for web browser extensions. Such extensions are available aplenty for Mozilla Firefox and Google Chrome and allow end-users to enrich their web browsing experience. Extensions contain code (both JavaScript and native code) that not only interacts with untrusted content on web pages, but also with code that accesses system resources, such as the file system and the network. It is critical to provide adversary containment for such extensions, e.g., to ensure that an adversary that hijacks an extension via a malicious script on a web page is unable to access system resources.

These problems have motivated much work from browser vendors in developing new frameworks for extension development [2, 8] that encourage extension developers to modularize their code. However, few guidelines exist for developers to understand how best to modularlize their code, and the process of creating modules is usually ad hoc. Moreover, browsers such as Firefox, which has only recently adopted modular extension development [2, 35], have legacy extensions that do not benefit from modularization.

We have applied modularization to legacy browser extensions, focusing first on Mozilla Firefox [34]. In this study, we transformed legacy extensions to benefit from the JetPack framework. We plan to extend this tool to also allow such extensions to operate atop Google Chrome, which uses different modularization primitives [71]. We also plan to apply our approach to traditional server software systems, such as the X server, OpenSSH, and the Apache web server.

6. RETROFIT FOR AUDITING

Retroactive security is the enforcement of security, or detection of security violations, after the execution of a process. By contrast, security mechanisms such as access control and information flow control enforce security either before or during execution. Years of experience with securing cyber systems has shown that retroactive security is necessary, in addition to protection-based mechanisms, since not all vulnerabilities can be predicted a priori or managed with prevention alone. Also, retroactive security can be used to hold entities accountable for their actions [44, 73, 72].

Auditing underlies retroactive security frameworks, and has become increasingly important to the theory and practice of cybersecurity and is essential for any defense in depth. However, auditing is error-prone, and difficult to get correct, in at least two ways. First, an audit log produced during execution must be an accurate record of all security-relevant events. Similar to missed access-control checks, it is easy for a programmer to accidentally omit the recording of all relevant events– for example, it has been shown that major health service informatics systems do not log sufficient information in light of guidelines for HIPAA policies [39]. Second, audit logs must be analyzed to detect security violations– a concern is often overlooked during development, resulting in "write only" logs that are never used for security enforcement.

Formal methods have been successful in addressing the second problem, and have been used to establish reliable foundations for analysis of audit logs [14, 67]. However, little attention has been paid to the first problem: assuring correctness of the audit log. Such assurances are essential for assuring any sort of security analysis based on auditing. Our goals here then are (1) to obtain a *semantics of audit logging* so that assurances can be meaningfully and rigorously obtained for auditing policies, and (2) to define policy-driven retrofitting tools for audit log generation, that provably respect the semantics of input logging policies.

An Illustrative Example. Consider a medical records system at a hospital. Some patients' records are marked as sensitive. To ensure that medical staff has timely access to patient information (e.g., accessing a patient's record when they are admitted to the emergency room), the system allows access to any record by medical staff. The system does not enforce access control restrictions, and allows medical staff to read from medical records, and send the record to others. To ensure that staff do not violate this trust and only use their access appropriately, the system should record in a log when a user reads and subsequently sends a sensitive medical record.

The medical records system must be instrumented to generate the appropriate logs. However, if instrumentation strategies are informal, then the intended policy may not be implemented. For example, developers may just "eyeball" the code to identify where a medical record may be sent and insert code to record this event in the audit log. But this strategy might record false positives if it is not statically known that a secure file is read, prior to the send. It can also lose information, since it may be difficult to statically predict sequences of function calls, especially in the presence of features such as dynamic dispatch.

Observe that the problem is not with the manner in which the audit log is queried, but rather with the way the audit log itself is generated. In particular, the instrumentation of the program does not properly realize the intended logging policy. Here is a more precise textual specification of what should be recorded in the log, which we call LP_H:

> LP_H : *Record in the log information associated with a remote send by a medical staff member, if a sensitive file was read by that staff member prior to the send.*

Subsequently, if system administrators discover that sensitive information is being leaked to some remote location, they may desire to ask the following audit query, which we call AQ_H:

> AQ_H : *Retrieve all destination addresses of remote sends by medical staff in the log file.*

However, note that while administrators expect that the answer to AQ_H will return e.g. *every* relevant potential recipient of sensitive information, this is the case only if LP_H has been instrumented correctly. If logging is incomplete, for example, then some potential recipients may be missed. If logging is overzealous, some legitimate recipients of sensitive information may be erroneously flagged. In other words, the connection between audit queries and log-generating processes is the manner in which programs are instrumented to generate logs, and correctness of logging instrumentation is vital for auditing assurances. By "instrumented", we mean the functionality that is added to code specifically to generate logs.

Approach Overview. We advocate for retrofitting approaches to auditing, since such tools can assure correct audit log generation even for untrusted code. That is, if retrofitting tools can be shown to correctly instrument *any* input program to support some class of logging policies, correctness of generated audit logs is automatically ensured. Clearly, a formal semantics of audit logging is necessary to establish correctness of retrofitting strategies.

We regard an audit log as a piece of information that is a refinement of the information contained in a process. Thus, the proper meaning of an auditing policy is as a kind of operation over information structures. With this view, it is natural to pursue a semantics of auditing based on *information algebra* [41, 40], which is a generalized framework for information systems. Information algebra has been shown to capture systems such as relational algebra, predicate logic, and linear systems. Aside from the philosophical appeal of realizing an auditing semantics in this general theory, an information algebra formulation has a number of technical advantages. For example, relations between distinct information algebras have been established, so the FOL-centric results in this paper can be ported to other systems, e.g. relational databases. Significantly, audit algebras enjoy a partial *information ordering*, denoted $\leq$ that allows comparison of information pieces wrt their information content. This ordering is crucial in relating audit logs with logging policy semantics, and establishing notions of soundness and completeness of retrofitting. Although the former is concrete, whereas the latter is abstract, they can be related by the information they contain.

In more detail, we argue that the semantics of a particular logging policy LP, which is specified in some formal language, be defined as an operation in a complete program trace. That is, for any program $\mathfrak{p}$, the semantics of a logging policy are a refinement of $\mathfrak{p}$'s complete execution trace, denoted $\mathit{traceof}(\mathfrak{p})$. This refinement can be specified as an information algebraic operation we call *genlog* that takes as input $\mathit{traceof}(\mathfrak{p})$ and LP, so that:

$$\mathit{genlog}(\mathit{traceof}(\mathfrak{p}), LP)$$

denotes the intended semantics of a logging policy LP for a given program $\mathfrak{p}$.

Given this semantic definition, as well as notions of information ordering available in information algebras, we can meaningfully define correctness of retrofitting strategies. Let $\mathbf{retro}$ be some retrofitting strategy, that takes as input programs $\mathfrak{p}$ and a logging policy LP, where we assume that LP is selected from some nonempty set of logging policies $\mathcal{P}$ that the strategy supports. We write:

$$\mathbf{retro}(\mathfrak{p}, LP) \rightsquigarrow \mathbb{L}$$

to denote that the log $\mathbb{L}$ is generated by executing the program $\mathfrak{p}'$ that results from instrumenting the program $\mathfrak{p}$ to support the logging policy LP. We say that $\mathbf{retro}$ is *sound* iff $\mathbb{L}$ represents a subset of information in $\mathit{genlog}(\mathit{traceof}(\mathfrak{p}), LP)$, and $\mathbf{retro}$ is *complete* iff $\mathit{genlog}(\mathit{traceof}(\mathfrak{p}), LP)$'s information content is contained in $\mathbb{L}$'s. More precisely, we have:

DEFINITION 1. *A retrofitting strategy* $\mathbf{retro}$ *is* sound for $\mathcal{P}$ *iff for all* $\mathfrak{p} \in \mathcal{L}$ *and* $LP \in \mathcal{P}$*, we have that* $\mathbb{L} \leq \mathit{genlog}(\mathfrak{p}, LP)$*, where* $\mathbf{retro}(\mathfrak{p}, LP) \rightsquigarrow \mathbb{L}$.

DEFINITION 2. *A retrofitting strategy* $\mathbf{retro}$ *is* complete for $\mathcal{P}$ *iff for all* $\mathfrak{p} \in \mathcal{L}$ *and* $LP \in \mathcal{P}$*, we have* $\mathit{genlog}(\mathfrak{p}, LP) \leq \mathbb{L}$*, where* $\mathbf{retro}(\mathfrak{p}, LP) \rightsquigarrow \mathbb{L}$.

In work so far, we have defined a language of logging policies based on first order logic (FOL), and have formalized a notion of program traces expressed as sets of temporally ordered FOL formulae. These definitions, along with additional constructions, obtain an information algebraic framework in which audit logging can be endowed with a semantics defined in terms of algebraic operations. We have also defined a retrofitting strategy for a core functional calculus that supports an interesting class of logging policies, the so-called surveillance policies. This strategy has been verified to be sound and complete, in the information algebraic sense described above, using the Coq framework.

In ongoing work, our immediate research targets include developing retrofitting strategies for realistic programming languages with correctness assurances, as well as type-directed optimizations. These optimizations will be based on our previous work on temporally-sensitive typing analyses [60].

7. VALIDATING DEFENSE IN DEPTH

Building on methods to validate security controls for authorization, containment, and auditing, we identify three advantages to reasoning about all three in unison. The first advantage is that we may be able to optimize the use of security controls by eliminating redundant controls. For example, authorization may reliably control all adversary flows from one module to another, which may eliminate the need to separate those modules. The second advantage is that the actual runtime use of the program may motivate changes in security controls that improve security. For example, logging downgraded data could show that the downgrading task is more common and more complex than envisioned, motivating changes in the retrofitting policies to enact more authorization and/or containment. Finally, the third advantage is that assurance can encompass all three controls, providing a comprehensive validation of enforcement. We plan to develop formal verification techniques to certify the correctness of retrofitting, similar to CompCert [12] and Vellvm [77, 78].

Thus, we propose a unified framework for retrofitting programs for defense in depth spanning all three security controls. Figure 1 shows the expected high-level design of the STRATA framework. In this task, we will explore methods for step two, *unified retrofitting*. This step receives retrofitting policies for each of the three security controls, plus the program code and optional feedback from deployed security controls. The output from this step is the program code retrofitted for authorization, containment, and auditing that satisfies the retrofitting policies and is optimal with respect to the costs of the controls.

Unified policy representation. A distinct benefit of designing a multi-layered security framework from the ground up is the opportunity to unify and synthesize policies across layers. For example, authorization and auditing policies can be synthesized to ensure auditing, and thus retroactive accountability, if certain authorization conditions are not met by actors. Such a policy was already suggested in Section 6. A unified policy representation, capturing properties at each layer, can be used to specify this.

Key to enabling unified retrofitting is a uniform denotation of retrofitting policies across STRATA levels. Authorization, containment, and auditing all refer to subjects (to be controlled), objects (that may be accessible to subjects or may have security requirements), program flows (control, data, and traces), and security policies. We argue that it would be beneficial to apply a single language to express retrofitting policies at each layer. One option is to express security controls in terms of automata. For example, *I/O automata* are labeled transition models for asynchronous concurrent systems [**?**]; they are typically used to describe the behavior of a system interacting with its environment. In I/O-automata-based models of monitoring, the system (node) to be monitored and the monitor are represented as I/O automata, with the input and output actions of each automaton representing their interaction with the environment and each other. Security policies are defined in terms of allowed or disallowed executions (traces). Using I/O automata, we can capture requirements on input (e.g., control of various subjects to that input), output (e.g., the impact of the operation on the security of the object), and trace effects (e.g., logging in particular states). Further, extensions to I/O automata have been proposed to represent probabilistic policies [**?**] and model cost [18], so this approach could capture a variety of whole-system enforcement scenarios. A remaining challenge is to ensure that I/O automata are at least translatable to resident policy languages at each STRATA layer.

Optimization via synthesized transformations. Policies specify the semantics of security mechanisms, but uniform policies will also enable the implementation of policies via program constructs that leverage connections between layers. As a retrofitting policy is defined to be a set of connections among a set of program constructs, the goal of this task is two-fold: (1) produce a single set of program constructs from the three control-specific sets and (2) produce a single set of connections among them from three control-specific sets. While the naive approach to union the three construct and connection sets to form a multi-control retrofitting policies can yield a solution if one exists, it may miss opportunities to find better solutions. For example, if the same constructs are identified for containment and authorization, then a sub-optimal solution that employs containment to isolate the constructs when authorization effectively blocks illegal data can be eliminated. We will explore automated analyses to identify such opportunities. For example, we will explore methods to identify such dominance relations across controls. Recall that programmers produce retrofitting policies interactively with STRATA, so such analyses must be meaningful to programmers. Ultimately, we would like programmers to "program" the retrofitting policy interactively with STRATA.

The problem of transformation takes a program, a retrofitting policy, and a cost function and produces a retrofitted program that satisfies the retrofitting policy for the minimal cost. For individual controls, the cost function focuses on only one dimension at a time, but since different controls apply different cost metrics we must consider transformation as a multi-dimensional optimization where the retrofitting policy implies a set of constraints.

Improving retrofitting policies continuously. The goal of continuous improvement is to use knowledge of how programs are actually run to reduce the risks taken by the trade-off of security with functional concerns proactively. To address this problem, we will leverage the unification of security controls to collect information for guiding improvements to the security controls themselves. The problem is analogous to auditing, except that rather than looking for intrusions we will try to estimate the risks introduced by security controls quantitatively to identify those most in need of revision. This is sometimes called *feedback* in the systems literature. For the auditing example on downgrading, we may estimate risk by the percentage of data to redact or number of decisions necessary to identify the data to redact. Using these quantitative metrics we may identify more (fewer) program constructs in need of control or eliminate (add) connections that are violated (satisfied) in practice, resulting in more (fewer) security controls. In addition, we will explore methods to make retrofitting changes based on such findings automatically, leading to agile retrofitting of programs as they execute.

Verifying transformations. Having retrofitted software for defense in depth, how can we show that the retrofitted system preserve the functionality of the original software system? In fact, we are interested in demonstrating that the behavior of the retrofitted system is a subset of the monolithic original, with omitted behaviors being those excluded by a security policy.

In general, proving correctness of program transformations is a difficult challenge, one that has remained open for several decades, cf., the quest to produce provably correct compilers and program optimizers. However, over the last few years, there have been impressive developments in this domain, thanks to advances in interactive theorem proving systems and SMT solvers.

We plan to build upon this line of research to build a verified retrofitting pipeline. One approach that we plan to explore is the use of Coq [13] to achieve the goal of verified transformations. Coq is an interactive proof assistant that allows co-development of program transformations, themselves expressed in Coq, together with their proofs of correctness. Transformations can be developed iteratively with their proofs, using the Coq system to debug the transformations or their proofs as errors are discovered. Thus, when the transformation has been fully specified, it is also accompanied with a machine-checkable proof of correctness. This approach was recently used in Vellvm [77, 78] to prove the correctness of several optimizations within LLVM. As has been noted in Section 6, we have already used Coq to verify a retrofitting strategy for auditing in preliminary work.

8. CONCLUSIONS

Even when programmers decide to add the security controls necessary to implement *defense in depth*, they face many practical challenges. First, Defenses are often added manually, using an ad hoc process. Second, each security control typically uses its own policies and mechanisms, so the manual process has to be repeated for each control. Third, it is difficult to prove that a manual deployment of security controls provides an advertised level of assurance. Recent work on methods to retrofit legacy code with security controls has begun to address some of these issues, but these methods still require significant manual effort, do not explicitly map security goals to program code, and they do not reason about multiple security controls. In this paper, we propose the STRATA framework for retrofitting legacy code for authorization, containment, and auditing security controls. The STRATA framework implements a comprehensive view of assurance, with an emphasis on *automated* and *interactive* tools that developers can use to identify site-level security goals, in terms of a *retrofitting policy*, and retrofit legacy code to enforce security policies in a manner that can be machine-verified for assurance. We show how security controls can be retrofit individually by STRATA and how STRATA enables optimization, continuous improvement, and assurance across multiple security controls. We show that by reasoning about defense in depth a variety of advantages can be obtained, including optimization, continuous improvement, and assurance across multiple security controls.

Acknowledgements

We acknowledge the anonymous reviewers of this paper. This material is based upon work supported by the National Science Foundation Grant No. CNS-1408880.

9. REFERENCES

[1] Ecmascript Harmony modules. http://wiki.ecmascript.org/doku.php?id=harmony:modules.
[2] The mozilla jetpack extension development framework. https://wiki.mozilla.org/Jetpack.
[3] Apache security controls and auditing. http://www.isaca.org/Journal/Past-Issues/2003/Volume-5/Pages/Apache-Security-Controls-and-Auditing.aspx, 2013.
[4] F.38. sepgsql. http://www.postgresql.org/docs/9.1/static/sepgsql.html, 2013.
[5] Linux audit-subsystem design documentation for kernel 2.6, version 0.1. http://www.uniforum.chi.il.us/slides/HardeningLinux/LAuS-Design.pdf, 2013.
[6] qmail home page. http://qmail.omnis.ch/top.html, 2013.
[7] J. P. Anderson. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, HQ Electronics Systems Division (AFSC), October 1972.
[8] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA*. The Internet Society, 2010.
[9] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, HQ Electronic Systems Division (AFSC), March 1976.
[10] D. Brumley and D. Song. PrivTrans: Automatically partitioning programs for privilege separation. In *Proceedings of the USENIX Security Symposium*, August 2004.
[11] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Task-structured probabilistic I/O automata. In *Proceedings of 8th International Workshop on Discrete Event Systems*, pages 207–214, 2006.
[12] J. Carter. Using GConf as an Example of How to Create an Userspace Object Manager. *2007 SELinux Symposium*, 2007.
[13] The COMPCERT project. http://compcert.inria.fr/.
[14] The Coq proof assistant. http://coq.inria.fr/, 2008. Version 8.1.
[15] A. Datta, J. Blocki, N. Christin, H. DeYoung, D. Garg, L. Jia, D. K. Kaynar, and A. Sinha. Understanding and protecting privacy: Formal semantics and principled audit mechanisms. In *7th International Conference on Information Systems Security*, pages 1–27, 2011.
[16] D. Dean, E. W. Felten, and D. S. Wallach. Java security: From hotjava to netscape and beyond. In *In Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, 1996.
[17] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.
[18] M. Dhawan, C. Shan, and V. Ganapathy. Enhancing JavaScript with transactions. In *Proceedings of ECOOP'12, the 26^{th} European Conference on Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science (LNCS)*, pages 383–408, Beijing, China, June 2012. Springer.
[19] P. Drábik, F. Martinelli, and C. Morisset. Cost-aware runtime enforcement of security policies. In *Proceedings of the 8th International Workshop on Security and Trust Management (STM 2012)*, pages 1–16, 2013.
[20] D.Walsh. Selinux/apache. http://fedoraproject.org/wiki/SELinux/apache.
[21] A. Edwards, T. Jaeger, and X. 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.
[22] J. Epstein and J. Picciotto. Trusting X: Issues in building Trusted X window systems -or- what's not trusted about X? In *Proceedings of the 14th Annual National Computer Security Conference*, Oct. 1991. A survey of the issues involved in building trusted X systems, especially of the multi-level secure variety.
[23] D. Ferraiolo and R. Kuhn. Role-based access control. In *In Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.
[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.
[25] M. Frank, J. M. Buhmann, and D. Basin. On the definition of role mining. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, SACMAT '10, pages 35–44. ACM, 2010.
[26] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux Security Modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 330–339, Nov. 2005.
[27] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Apr. 1982.
[28] C. Grier, S. Tang, and S. King. Secure web browsing with the op web browser. In *IEEE Symposium on Security and Privacy*, 2008.
[29] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416. IEEE Computer Society, 2008.
[30] W. R. Harris, S. Jha, and T. W. Reps. DIFC programs by automatic instrumentation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 284–296, 2010.
[31] W. R. Harris, S. Jha, T. W. Reps, J. Anderson, and R. N. M. Watson. Declarative, temporal, and practical programming with capabilities. In *IEEE Symposium on Security and Privacy*, pages 18–32, 2013.
[32] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)*, December 2006.
[33] JDBC: Java Database Connectors. http://java.sun.com/products/jdbc.
[34] P. A. Karger and R. R. Schell. MULTICS security evaluation: Vulnerability analysis. Technical Report ESD-TR-74-193, Deputy for Command and Management Systems, Electronics Systems Division (ASFC), June 1974.
[35] R. Karim, M. Dhawan, and V. Ganapathy. Refactoring legacy browser

extensions to modern extension frameworks. In *Proceedings of ECOOP'14, the 28^{th} European Conference on Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science (LNCS)*, pages 463–488, Uppasala, Sweden, July/August 2014. Springer.

[36] R. Karim, M. Dhawan, V. Ganapathy, and C. Shan. An analysis of the Mozilla Jetpack extension framework. In *Proceedings of ECOOP'12, the 26^{th} European Conference on Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science (LNCS)*, pages 333–355, Beijing, China, June 2012. Springer.

[37] D. Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the 2003 USENIX Annual Technical Conference—FREENIX Track*, June 2003.

[38] D. H. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit Flows: Can't live with 'em, can't live without 'em. In *Proceedings of Fourth International Conference on Information Systems Security*, Dec. 2008.

[39] D. H. King, S. Jha, D. Muthukumaran, T. Jaeger, S. Jha, and S. Seshia. Automating security mediation placement. In *Proceedings of the 19th European Symposium on Programming (ESOP '10)*, pages 327–344, 2010.

[40] J. T. King, B. Smith, and L. Williams. Modifying without a trace: General audit guidelines are inadequate for open-source electronic health record audit mechanisms. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, IHI '12, pages 305–314, New York, NY, USA, 2012. ACM.

[41] J. Kohlas. *Information Algebras: Generic Structures For Inference*. Discrete mathematics and theoretical computer science. Springer, 2003.

[42] J. Kohlas and J. Schmid. An algebraic theory of information: An introduction and survey. *Information*, 5(2):219–254, 2014.

[43] A. Kurmus, R. Tartler, D. Dorneau, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schoeder-Preikschat, D. Lohman, and R. Kapitza. Attack surface metrics and automated compile-time os kernel tailoring. In *NDSS*, 2013.

[44] B. W. Lampson. Protection. In *5th Princeton Conference on Information Sciences and Systems*, 1971.

[45] B. W. Lampson. Computer security in the real world. *IEEE Computer*, 37(6):37–46, 2004.

[46] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *POPL*, pages 385–398, 2013.

[47] B. Livshits and J. Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *Proceedings of the 22nd USENIX Security Symposium*, Berkeley, CA, USA, 2013. USENIX Association.

[48] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings ACM Symposium on Principles of Distributed Computing*, 1987.

[49] I. Molloy, N. Li, T. Li, Z. Mao, Q. Wang, and J. Lobo. Evaluating role mining algorithms. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 95–104. ACM, 2009.

[50] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging 'choice' in authorization hook placement. In *19th ACM Conference on Computer and Communications Security*. ACM, 2012.

[51] D. Muthukumaran, T. Jaeger, and G. Tan. Producing hook placements to enforce expected authorization policies. Technical Report NSRC Technical Report NAS-TR-169-2013, The Pennsylvania State University, September 2013.

[52] D. Muthukumaran, J. Schiffman, M. Hassan, A. Sawani, V. Rao, and T. Jaeger. Protecting the integrity of trusted applications on mobile phone systems. *Security and Communication Networks*, 4(6), 2011.

[53] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. Adsafety: type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 12–12. USENIX Association, 2011.

[54] The Postfix mail program. http://www.postfix.org.

[55] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[56] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *HotBots*, 2007.

[57] C. Reis, A. Barth, and C. Pizano. Browser security: Lessons from Google Chrome. *Communications of the ACM*, 52(8):45–49, Aug. 2009.

[58] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. nald Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based security architecture for the xen open-source hyperviso r. In *Proceedings of the 2005 Annual Computer Security Applications Conference*, pages 276–285, Dec. 2005.

[59] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[60] Re: Adding support for SE-Linux security. http://archives.postgresql.org/pgsql-hackers/2009-12/msg00735.php, 2009.

[61] SE-PostgreSQL? http://archives.postgresql.org/message-id/20090718160600.GE5172@fetter.org, 2009.

[62] C. Skalka, S. Smith, and D. V. Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2):179–249, 2008.

[63] S. Son, K. S. McKinley, and V. Shmatikov. 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*, OOPSLA '11, pages 1069–1084. ACM, 2011.

[64] S. Son, K. S. McKinley, and V. Shmatikov. Fix Me Up: Repairing Access-Control Bugs in Web Applications. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[65] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov. A security policy oracle: detecting security holes using multiple api implementations. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 343–354. ACM, 2011.

[66] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 11–11. USENIX Association, 2011.

[67] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: automatically inferring security specifications and detecting violations. In *Proceedings of the 17th conference on Security symposium*, pages 379–394. USENIX Association, 2008.

[68] D. Turner. Symantec internet security threat report: Trends for january-june 07. Technical report, Symantec Inc., 2007.

[69] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *In Proc. of the IEEE Computer Security Foundations Symposium*, 2008.

[70] E. Walsh. Integrating x.org with security-enhanced linux. In *Proceedings of the 2007 Security-Enhanced Linux Workshop*, Mar. 2007.

[71] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*, February 2006.

[72] R. N. M. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 15–28. USENIX Association, 2001.

[73] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security*, 2010.

[74] D. J. Weitzner. Beyond secrecy: New privacy protection strategies for open information spaces. *IEEE Internet Computing*, 11(5):94–96, 2007.

[75] D. J. Weitzner, H. Abelson, T. Berners-Lee, J. Feigenbaum, J. A. Hendler, and G. J. Sussman. Information accountability. *Communications of the ACM*, 51(6):82–87, 2008.

[76] C. Wright, C. Cowan, and J. Morris. Linux Security Modules: General security support for the Linux kernel. In *In Proceedings of the 11th USENIX Security Symposium*, pages 17–31, 2002.

[77] Implement keyboard and event security in X using XACE. https://dev.laptop.org/ticket/260, 2006.

[78] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, August 2002.

[79] J. Zhao, S. Nagarakatte, M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *ACM Principles of Programming Languages (POPL)*, 2012.

[80] J. Zhao, S. Nagarakatte, M. Martin, and S. Zdancewic. Formal verification of SSA optimizations. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013.