2016 IEEE 29th Computer Security Foundations Symposium

On access control, capabilities, their equivalence, and confused deputy attacks

Vineet Rajani
MPI-SWS

Deepak Garg
MPI-SWS

Tamara Rezk
INRIA

***Abstract*—Motivated by the problem of understanding the difference between practical access control and capability systems formally, we distill the essence of both in a language-based setting. We first prove that access control systems and (object) capabilities are fundamentally different. We further study capabilities as an enforcement mechanism for confused deputy attacks (CDAs), since CDAs may have been the primary motivation for the invention of capabilities. To do this, we develop the first formal characterization of CDA-freedom in a language-based setting and describe its relation to standard information flow integrity. We show that, perhaps suprisingly, capabilities cannot prevent all CDAs. Next, we stipulate restrictions on programs under which capabilities ensure CDA-freedom and prove that the restrictions are sufficient. To relax those restrictions, we examine provenance semantics as sound CDA-freedom enforcement mechanisms.**

***Keywords*-Access control; Capability; Confused deputy problem; Provenance tracking; Information flow integrity**

I. Introduction

Access control and capabilities are the most popular mechanisms for implementing authorization decisions in systems and languages. Roughly, whereas in access control a token (authentication credential) that represents the current principal is associated to a list of authorization rights, in capabilities the authorization right is the token (capability). Although, both mechanisms have been widely studied and deployed at various levels of abstraction (see e.g. [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]), there seems to be no clear consensus on the fundamental difference in their modus operandi. Our broad goal is to formalize fundamental properties that distinguish access control and capability systems from each other. Our motivation is partly pedagogic, and partly to discover the limits of what can and cannot be enforced using access control systems and capabilities. Against the backdrop of this broad goal, we make three contributions in this paper.

First, we reflect upon the question of whether or not access control and capability systems are equivalent in a formal language-based setting. Specifically, we are interested in this question in the context of capability systems that possess what Miller *et al.* [11] call Property A or "no designation without authority": If a principal acquires a capability, it also acquires the authority to use it.[1] Property A is very interesting because it is fundamental to many types of capability systems including all object capability systems, which are used to obtain isolation and security in large code bases [5], [3], [2]. In an object capability system, a capability is a reference to an ordinary language object or a memory location and there are no checks on using references, so the possessor of a capability can always read or write it.

To formalize access control and capabilities, we design a small core calculus with regions (principals) and memory references (objects/capabilities), and equip it with two different semantics—an access control semantics and a capability semantics with property A. We then show that access control and capabilities with property A are fundamentally different: The access control semantics is strictly more permissive than the capability semantics. (This formally justifies an earlier *informal* argument to the same effect by Miller *et al.* [11].)

Our access control semantics is the expected one. It intervenes on every use (read/write) of a reference and checks that the use is compliant with a given access policy. The capability semantics is less obvious, so we briefly describe its design here. By contraposition of the definition of property A, it follows that to limit authority in a capability system with property A, we must limit the designation of capabilities. In general, principals may acquire capabilities either by generating them (e.g., by guessing them or computing them from existing values) or by receiving them from other principals. Hence, to get security, i.e., to control authority, a capability system must ensure that:

1) A principal cannot generate a capability he is not authorized to use, and
2) A principal cannot receive (from another principal) a capability he is not authorized to use.

In practical systems with property A, (1) is ensured by using abstract, unforgeable tokens for capabilities. (1) is closely related to, and usually implied by, a property called "capability safety" [12]. Capability safety requires that a principal may acquire a capability only if the capability, as an object, is reachable in the initial heap starting from the principal's initial set of capabilities. So capability safety immediately implies (1). However, (1) in itself is not enough to get security; we also need (2).

How do we enforce (2)? One option is to *define* authority as the set of all capabilities that are obtained during program execution. Then, (2) holds trivially. However, this implicit definition of authority allows bugs in the code to leak

[1]The word principal should be interpreted broadly here. It may refer to a section of code, a function, or a user.

DOI 10.1109/CSF.2016.18

capabilities that the programmer never intended, without breaking allowed authority in a formal sense.

An alternative to this implicit specification of authority is to specify, via an explicit access policy, what references (capabilities) each principal is authorized to access and to ensure that each principal can only obtain references that it can legitimately use. This approach is taken in some practical implementations of object capabilities, e.g., Firefox's security membrane [5], which intercepts all transmissions from one domain to another and restricts objects (capabilities) in accordance with relevant policies (Firefox's policies are drawn from web standards like the same-origin policy). Our capability semantics models a simplified version of this general pattern. It intervenes on every transmitted and computed value and checks that if the value is a reference (capability), then the executing principal is authorized to use the capability according to the access policy. This intervention is computationally expensive, since every value must be checked. Nonetheless, our capability semantics is an ideal model of how security is enforced in the presence of property A and an explicit access policy. Interestingly, our semantics enforces not just (2) but also (1), so there is no need to make capabilities unforgeable. Hence, our approach is compatible with a language that includes pointer arithmetic.[2]

As a second contribution, we formally examine confused deputy attacks (CDAs) [16], which may have been the primary reason for the invention of capabilities. A CDA is a privilege escalation attack where a deputy (a trusted system component) can act on both its own authority and on an adversary's authority. In a CDA, the deputy is confused because it thinks that it is acting on its own authority when in reality it is acting on an attacker's authority. Cross-site request forgery [17], FTP bounce attacks [18] and clickjacking [19] are all prevalent examples of CDAs. It is widely known that access control alone is insufficient to prevent CDAs and it is known that the use of capabilities prevents (at least some) CDAs.

We make two fundamental contributions in the context of CDAs. First, we provide what we believe to be the first formal definition of when a program is free from CDAs. Our definition is extensional and is inspired by information flow integrity [20], [21], [22], but we show that CDA-freedom is strictly weaker than information flow integrity. Second, we use this definition and our capability semantics to formally establish that, perhaps surprisingly, capability semantics is not enough to ensure CDA-freedom. While capabilities prevent many CDAs that are based on explicit designation of authority from the adversary to the deputy, there are other CDAs based on implicit designation that capability semantics cannot prevent. We also stipulate restrictions on programs under which capability semantics prevent all CDAs. However, these restrictions are very strong and render the language useless for almost all practical purposes.

As our final contribution, we investigate alternate approaches for CDA prevention with fewer restrictions. Our approaches rely on provenance tracking (taint tracking). First, we formally show that merely tracking *explicit* provenance (i.e., without taking into account influence due to control flow) suffices to guarantee CDA-freedom with fewer restrictions than capabilities require. In order to remove even these restrictions, we further develop a full-fledged provenance analysis and prove CDA-freedom. We compare the three methods of preventing CDAs (capabilities, explicit provenance tracking, full provenance tracking) in terms of permissiveness through examples.

To summarize, the key contributions of this work are:

- We formally examine the fundamental difference between access control and capabilities in a language-based setting.
- We give the first extensional characterization of CDA-freedom and its relation to information-flow integrity.
- We show that capability semantics are not enough for CDA-freedom in the general case. We then examine conditions under which this implication holds.
- We present provenance tracking as an alternate approach for preventing CDAs with fewer assumptions and prove its soundness.

Proofs and many other technical details are available in a technical report available from the authors' homepages. The technical report also considers an extension of our calculus with computable references (pointer arithmetic).

II. Access control vs Capabilities

Our technical development is based on a region calculus, a simple, formal imperative language with notions of principals (which own a subset of references) and regions (which specify a write integrity policy that we wish to enforce). This simple calculus suffices to convey our key ideas, without syntactic clutter. The syntax of our calculus is shown in Figure 1. We assume two countable sets, Loc of mutable references and $Prin$ of principals. Elements of Loc are written r and elements of $Prin$ are written $\mathbb{P}$. Our calculus has five syntactic categories — values (v), expressions (e), commands (c), regions (ρ), and top-level programs or, simply, programs (P).

Values consist of integers (n), booleans (tt, ff) and pointers or mutable references ${}^{\mathbb{R}}r$ and ${}^{\mathbb{W}}r$. References ${}^{\mathbb{R}}r$ and ${}^{\mathbb{W}}r$ represent the read and write capabilities for the reference r. Capability ${}^{\mathbb{R}}r$ can only be used to read r, whereas capability ${}^{\mathbb{W}}r$ can only be used to write r. Separating these capabilities allows us to make a fine distinction between security checks

[2]Going beyond policy enforcement is the question of whether the policy attains higher-level security goals such as ensuring specific invariants on protected state or limiting observable effects to a desirable set. Such higher-level goals can be attained using static verification, as in [13], [14], [15], but these goals are beyond the scope of this paper.

$v ::=$		Value
	n	Integer
	tt	True
	ff	False
	$^{\mathbb{R}}r$	Read view of a location
	$^{\mathbb{W}}r$	Write view of a location
$e ::=$		Expression
	v	Value
	$!e$	Dereference
$c ::=$		Command
	if e then c else c	Conditional
	while e do c	Loop
	$e := e$	Assignment
	$c; c$	Sequential composition
	skip	Skip
$\rho ::=$		Region
	$\mathbb{P}$	Principal
	$\overline{\mathbb{P}}$	Endorsed principal
$P ::=$		Program
	$\rho\{c\}$	Region command
	$P \circ P$	Region composition

Figure 1. Region calculus

on reads and writes. Expressions e are computations that cannot update references. They include values and reference reading ($!e$). Commands c are standard conditionals, while loops, assignments, sequencing ($c_1; c_2$) and skip.

A region ρ is either a principal $\mathbb{P}$ or an *endorsed principal*, $\overline{\mathbb{P}}$. In both cases, $\mathbb{P}$ represents a ceiling (maximum) authority for executing code. However, in the case of an endorsed region, the principal expresses the explicit willingness to act on another principal's behalf. In our definition of CDA-freedom (Definition 3), we take this intention into account to explicitly exclude endorsed regions as sources of CDAs. For now, readers may ignore endorsed principals $\overline{\mathbb{P}}$, treating them exactly like normal principals $\mathbb{P}$.

A program P is a sequence of commands, executed in possibly different regions. A program has the form $\rho_1\{c_1\} \circ \ldots \circ \rho_n\{c_n\}$ and means that first command c_1 runs in the region ρ_1, then command c_2 runs in region ρ_2 and so on. When a command runs in a region, the command is subject to the ceiling authority of the region.

Regions and write integrity: The primary property we wish to enforce is *write integrity*. To specify this property, we assume that each reference is *owned* by a principal. This is formalized by an ownership map $\mathbb{O}$, that maps a reference to the principal that owns the reference. Formally, $\mathbb{O} : Loc \rightarrow Prin$. Principals are assumed to be organized in a lattice $\mathbb{L}$ whose order is written $\geq_{\mathbb{L}}$. This lattice is a technical representation of a write integrity policy: Code executing in region $\mathbb{P}$ or $\overline{\mathbb{P}}$ can write to reference r, i.e., it can wield the capability $^{\mathbb{W}}r$ only if $\mathbb{P} \geq_{\mathbb{L}} \mathbb{O}(r)$.[3] For convenience, we extend the order $\geq_{\mathbb{L}}$ to regions: $\rho \geq_{\mathbb{L}} \rho'$ when $\rho \in \{\mathbb{P}, \overline{\mathbb{P}}\}$, $\rho' \in \{\mathbb{P}', \overline{\mathbb{P}'}\}$ and $\mathbb{P} \geq_{\mathbb{L}} \mathbb{P}'$.

It should be clear that the lattice $\mathbb{L}$ and the ownership map $\mathbb{O}$ together define an access/authorization policy for write references. We enforce this policy using either access control or capability-based checks, as explained below. Authorization for read references is also important in practice, but is not the focus of this paper. In fact, we allow any command to dereference any read capability the command possesses.

Access control and capability semantics: Since our first goal is to investigate the differences between access control systems and capability systems, we equip our calculus with two different runtime semantics — an access control semantics (ACs) and a capability semantics (Cs). Both enforce write integrity, but in different ways. Whereas ACs checks that the ceiling authority is sufficient when a reference is written (through the policy described above), Cs prevents a region from getting a write capability which it cannot wield in the first place. Technically, Cs must intercept every constructed value and check that, if the value is a write capability, then the executing region is higher (in $\mathbb{L}$) than the region that owns the reference accessible through the capability. While this is cumbersome, in our opinion, this is the formal essence of Miller *et al.*'s Property A of capability systems [11]: possession of a reference (capability) implies the authority to use it. By inference, if a region must not write a reference according to the policy, it must not ever possess the reference. We now formalize the two semantics ACs and Cs.

As usual, a heap H is a map from Loc to values and determines the value stored in each reference. Here, values are integers and booleans. Both ACs and Cs are defined by three evaluation judgments: $\langle H, e\rangle \overset{\rho}{\Downarrow_X} v$ for expres-

[3]The lattice specifies only an upper bound or ceiling on the set of references the code in a region can write. However, the code must also explicitly present a write capability to a reference in order to update the reference. Miller *et al.* call this requirement to explicitly present capabilities "property D" or "no ambient authority", and argue that it is a pre-requisite for ruling out confused-deputy attacks [11].

sions, $\langle H, c\rangle \xrightarrow{\rho}_X \langle H', c'\rangle$ for commands and $\langle H, P\rangle \rightarrow_X \langle H', P'\rangle$ for programs. Here X may be A (for the semantics ACs) or C (for the semantics Cs). In the rules for expressions and commands, ρ denotes the region or the ceiling authority in which evaluation happens. Figure 2 shows all the semantic rules. When a rule applies to both ACs and Cs, we use the generic index X in both the name of the rule and on the reduction arrow.

The judgment for expression evaluation $\langle H, e\rangle \overset{\rho}{\Downarrow}_X v$ means that when the heap is H, expression e evaluates to value v. The ACs rules (top of Figure 2, left panel) are straightforward. For dereferencing, we need the read capability ${}^{\mathbb{R}}r$ (rule A-Deref). The Cs rules (right panel) are exactly like the access control rules, but they all make an additional check: If the value being returned is a write capability, then the executing region ρ must be above the owner of the capability's reference. This ensures that the executing region never gets a write capability whose owner's authority is not below the executing region's authority.

The judgment for command evaluation $\langle H, c\rangle \xrightarrow{\rho}_X \langle H', c'\rangle$ means that c reduces (one-step) to c' transforming the heap from H to H'. The rules for this judgment are mostly standard. The only interesting point is that in the ACs rule for reference update (rule A-Assign), we check that the owner $\mathbb{O}(r)$ of the updated reference r is below the executing region ρ. A corresponding check is not needed in Cs (rule C-Assign) because, there, the assigned reference r cannot even be computed unless $\rho \geq_{\mathbb{L}} \mathbb{O}(r)$. Technically, the rules for $\overset{\rho}{\Downarrow}_C$ ensure that the check $\rho \geq_{\mathbb{L}} \mathbb{O}(r)$ is made in the derivation of the first premise of C-Assign.

Access control more permissive than capabilities: We now prove that ACs is strictly more permissive than Cs, thus accomplishing our first goal. The extra permissiveness of ACs over Cs should be expected because Cs prevents code from obtaining write capabilities that it cannot use whereas ACs allows the region to obtain such capabilities, but prevents it from writing them (later). The following theorem formalizes this intuition. It says that if a reduction is allowed in Cs, then the reduction must also be allowed in ACs.

Theorem 1 (ACs more permissive than Cs)**.** $\langle H, P\rangle \rightarrow_C \langle H', P'\rangle$ implies $\langle H, P\rangle \rightarrow_A \langle H', P'\rangle$.

Proof: By induction on the given derivation of $\langle H, P\rangle \rightarrow_C \langle H', P'\rangle$. ■

The converse of this theorem is false. For example, consider the program $\rho\{{}^{\mathbb{W}}r_1 := {}^{\mathbb{W}}r_2\}$ that runs with ceiling authority ρ and stores the reference r_2 in the reference r_1. Assume that $\rho \geq_{\mathbb{L}} \mathbb{O}(r_1)$ but $\rho \not\geq_{\mathbb{L}} \mathbb{O}(r_2)$, i.e., ρ can write to r_1 but not to r_2. Then, ACs allows the program to execute to completion. On the other hand, Cs blocks this program because ρ will not be allowed to compute the capability ${}^{\mathbb{W}}r_2$, which it cannot wield. Technically, the second premise of rule C-Assign will not hold for this example. Hence, the access control semantics, ACs, is strictly more permissive than the capability semantics, Cs.

Write integrity: Despite their differences, both ACs and Cs provide write integrity in the sense that neither allows a region to write a reference that it is not authorized to write. We formalize and prove this result below.

Theorem 2 (ACs and Cs provide write integrity)**.** If $\langle H, \rho\{c\}\rangle \rightarrow_X{}^* \langle H', \_\rangle$ and $H(r) \neq H'(r)$, then $\rho \geq_{\mathbb{L}} \mathbb{O}(r)$.

Proof: For $X = \text{A}$, the result is proved by induction on the reduction sequence $\rightarrow_A^*$ and, at each step, by induction on the derivation of the given reduction. For $X = \text{C}$, the result follows from the result for $X = \text{A}$ and Theorem 1. ■

Capability Safety: Capability safety is a widely discussed, but seldom formalized foundational property of capability-based languages. Roughly, it says that capabilities to access resources can only be obtained through legal delegation mechanisms. We have proved capability safety for Cs by instantiating a general definition of the property due to Maffeis *et al.* [12]. Since capability safety is largely orthogonal to our goals, we relegate its details to our technical report.

Theorem 3 (Capability Safety)**.** The semantics Cs is capability safe.

III. Confused Deputy Attacks and Capabilities

A confused deputy attack [16], CDA for short, is a privilege escalation attack where the adversary who doesn't have direct access to some sensitive resource, indirectly writes the resource by confusing a deputy, a principal who can access the resource. The confused compiler service is a folklore example of a CDA. In this example, a privileged compiler service is tricked by its unprivileged caller into overwriting a sensitive billing file which the caller cannot update, but the compiler can. The compiler service takes as inputs the names of the source file to be compiled and an output file. It compiles the source file, writes the compiled binary to the output file and, importantly, on the side, writes a billing file that records how much the caller must pay for using the compiler. The caller tricks the compiler by passing to it the name of the billing file in place of the output file, which causes the compiler to overwrite the billing file with a binary, thus destroying the billing file's integrity. (Of course, pay-per-use compilers are rare today, but the example is very illustrative and CDAs remain as relevant as ever.)

CDAs are interesting from our perspective because they distinguish access control semantics, which offer no defense against CDAs from capability semantics, which can prevent at least some CDAs. For instance, Cs would prevent the CDA in the compiler service example above, but ACs would not (see later examples for a proof). It has been claimed in the past that in the presence of Miller *et al.*'s property A, Cs

Expressions:

Access control

$$\text{A-Val}\ \frac{}{\langle H, v\rangle \overset{\rho}{\Downarrow}_A v}$$

$$\text{A-Deref}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_A {}^{\mathbb{R}}r \qquad v = H(r)}{\langle H, !e\rangle \overset{\rho}{\Downarrow}_A v}$$

Capability

$$\text{C-Val}\ \frac{v = {}^{\mathbb{W}}r \implies \rho \geq_{\mathbb{L}} \mathbb{O}(r)}{\langle H, v\rangle \overset{\rho}{\Downarrow}_C v}$$

$$\text{C-Deref}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_C {}^{\mathbb{R}}r \qquad v = H(r) \qquad v = {}^{\mathbb{W}}r' \implies \rho \geq_{\mathbb{L}} \mathbb{O}(r')}{\langle H, !e\rangle \overset{\rho}{\Downarrow}_C v}$$

Commands:

$$\text{X-if}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_X v \qquad v = tt}{\langle H, \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \overset{\rho}{\rightarrow}_X \langle H, c_1\rangle} \qquad \text{X-else}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_X v \qquad v = ff}{\langle H, \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \overset{\rho}{\rightarrow}_X \langle H, c_2\rangle}$$

$$\text{X-while 1}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_X v \qquad v = tt}{\langle H, \text{while } e \text{ do } c\rangle \overset{\rho}{\rightarrow}_X \langle H, c; \text{while } e \text{ do } c\rangle} \qquad \text{X-while 2}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_X v \qquad v = ff}{\langle H, \text{while } e \text{ do } c\rangle \overset{\rho}{\rightarrow}_X \langle H, \text{skip}\rangle}$$

Access control

$$\text{A-Assign}\ \frac{\langle H, e_1\rangle \overset{\rho}{\Downarrow}_A {}^{\mathbb{W}}r \qquad \rho_r = \mathbb{O}(r) \qquad \rho \geq_{\mathbb{L}} \rho_r \qquad \langle H, e_2\rangle \overset{\rho}{\Downarrow}_A v}{\langle H, e_1 := e_2\rangle \overset{\rho}{\rightarrow}_A \langle H[r \mapsto v], \text{skip}\rangle}$$

Capability

$$\text{C-Assign}\ \frac{\langle H, e_1\rangle \overset{\rho}{\Downarrow}_C {}^{\mathbb{W}}r \qquad \langle H, e_2\rangle \overset{\rho}{\Downarrow}_C v}{\langle H, e_1 := e_2\rangle \overset{\rho}{\rightarrow}_C \langle H[r \mapsto v], \text{skip}\rangle}$$

$$\text{X-Seq 1}\ \frac{\langle H, c_1\rangle \overset{\rho}{\rightarrow}_X \langle H', c_1'\rangle}{\langle H, c_1; c_2\rangle \overset{\rho}{\rightarrow}_X \langle H', c_1'; c_2\rangle} \qquad \text{X-Seq 2}\ \frac{}{\langle H, \text{skip}; c_2\rangle \overset{\rho}{\rightarrow}_X \langle H, c_2\rangle}$$

Program:

$$\text{X-Prg 1}\ \frac{\langle H, c\rangle \overset{\rho}{\rightarrow}_X \langle H', c'\rangle}{\langle H, \rho\{c\}\rangle \rightarrow_X \langle H', \rho\{c'\}\rangle} \qquad \text{X-Comp 1}\ \frac{\langle H, P_1\rangle \rightarrow_X \langle H', P_1'\rangle}{\langle H, P_1 \circ P_2\rangle \rightarrow_X \langle H', P_1' \circ P_2\rangle}$$

$$\text{X-Comp 2}\ \frac{}{\langle H, \rho\{\text{skip}\} \circ P\rangle \rightarrow_X \langle H, P\rangle}$$

Figure 2. Access control (A) and Capability (C) semantics

prevent CDAs [16], [11], [23]. However, to the best of our knowledge, there is, thus far, no formal characterization of what it means for a system to be free from a CDA, nor a formal understanding of whether all CDAs can be prevented by Cs. In this section, we address both these issues. First, we provide a formal definition of what it means for a program to be free from CDAs (subsection III-A). Then we show that Cs cannot prevent all CDAs even in a minimalist language such as our region calculus, but they can actually prevent all CDAs under very strong restrictions (subsection III-B). This provides the first formal characterization of a language (fragment) in which capabilities can provably prevent CDAs.

A. Defining CDA-freedom

The goal of this subsection is to define what it means for a program to be CDA-free. To test whether a program is free from CDAs or not, the program must allow for interaction with an adversary. To this end, we define an *authority context* or, simply, context, written $\mathbb{E}_{\rho_A}$, which is a program with one hole, where an adversary's commands can be inserted. We write ρ_A for the adversary region that has a hole.

Definition 1 (Authority Context)**.** An authority context, $\mathbb{E}_{\rho_A}$, is a program with one hole of the form $\rho_A\{\bullet\}$. Formally, $\mathbb{E}_{\rho_A} ::= \rho_1\{c_1\} \circ \ldots \circ \rho_A\{\bullet\} \circ \ldots \circ \rho_n\{c_n\}$. We write $\mathbb{E}_{\rho_A}[c_A]$ for the program that replaces the hole $\bullet$ with the

adversary's commands c_A, i.e., the program $\rho_1\{c_1\} \circ \ldots \circ \rho_A\{c_A\} \circ \ldots \circ \rho_n\{c_n\}$.

Any program P (without a hole) can be trivially treated as an authority context $\mathbb{E}_{\rho_A} = P \circ \rho_A\{\bullet\}$. In some examples, we treat programs as authority contexts in this sense.

In a CDA, the goal of an adversary is to overwrite one or more references. We call these references the "attacker's interest set", denoted AIS. In the sequel, we assume a fixed AIS. Intuitively, a context $\mathbb{E}_{\rho_A}$ is free from a CDA if for every reference $r \in AIS$ either the attacker cannot control what value is written to r, or the attacker can write to r directly. If the first disjunct holds, then there is no attack on r whereas if the second disjunct holds, then there is no need for a confused deputy (the context $\mathbb{E}_{\rho_A}$) to modify r. In either case, there is no confused deputy attack on r. The first disjunct can be formalized by saying that no matter what adversary code we substitute into $\mathbb{E}_{\rho_A}$'s hole, the final value in r is the same. The second disjunct can be formalized by saying that there must be *some* adversary code that, when running with the ceiling authority ρ_A, can write the final value of r to r directly. Based on this, we arrive at the following preliminary definition of CDA-freedom (we revise this definition later). Here, `final` denotes a fully reduced program, of the form $\rho\{\text{skip}\}$.

Definition 2 (CDA-freedom)**.** Context $\mathbb{E}_{\rho_A}$ starting from the initial heap H and running under reduction semantics $\rightarrow_{red}$ is said to be free from CDAs, written $\text{CDAF}(\mathbb{E}_{\rho_A}, H, \rightarrow_{red})$, if for every c_A and H' such that $\langle H, \mathbb{E}_{\rho_A}[c_A]\rangle \rightarrow^*_{red} \langle H', \texttt{final}\rangle$ and for every $r \in AIS$ at least one of the following holds:

1) (No adversary control) For any c'_A, it is the case that if $\langle H, \mathbb{E}_{\rho_A}[c'_A]\rangle \rightarrow^*_{red} \langle H'', \texttt{final}\rangle$ then $H'(r) = H''(r)$, or
2) (Direct adversary write) There exists a c''_A such that $\langle H, \rho_A\{c''_A\}\rangle \rightarrow^*_{red} \langle H''', \texttt{final}\rangle$ and $H'(r) = H'''(r)$.

Note that this definition does not require that the same clause (1 or 2) hold for every $r \in AIS$. Instead, some r may satisfy clause 1 and others may satisfy clause 2. This definition is inspired by and strictly weaker than information flow integrity (we show this formally in Section V).

Example 1 (Compiler service, simplified)**.** We formalize a simplified version of the confused compiler service described at the beginning of this section. The simplification is that this compiler does not contain the code that writes the billing file (we add the billing file in Example 5). Suppose that the compiler runs with authority $\top$ (the highest authority), the compiler service's caller/adversary runs with authority $\bot$ (the lowest authority, $\bot \not\geq_{\mathbb{L}} \top$) and that the compiler reads the source program from the reference r_S and the *name* of the output file from the reference r_O, both of which the caller must write beforehand. Then, we can abstractly model the relevant parts of the compiler service as the context $\mathbb{E}_\bot = \bot\{\bullet\} \circ \top\{(!^{\mathbb{R}} r_O) := \texttt{compile}(!^{\mathbb{R}} r_S)\}$, where `compile` compiles a program. Note that this program has a CDA, i.e., it is not CDA-free according to Definition 2. For instance, consider adversaries of the form $c_A(S) = ({}^{\mathbb{W}} r_O := {}^{\mathbb{W}} r; {}^{\mathbb{W}} r_S := S)$, where $r \in AIS$ is a reference with $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r)$ and S ranges over source programs. Consider the execution of $\mathbb{E}_\bot[c_A(s_1)]$ for some source program s_1. Then, clause (1) does not hold for r because for another source program s_2 with $\texttt{compile}(s_1) \neq \texttt{compile}(s_2)$, the final heaps from the executions of $\mathbb{E}_\bot[c_A(s_1)]$ and $\mathbb{E}_\bot[c_A(s_2)]$ disagree on r. Clause (2) clearly does not hold when the initial heap does not contain $\texttt{compile}(s_1)$ in r. Intuitively, the CDA here is the expected one: The adversary passes whatever reference it wishes to overwrite in place of the output file.

In this case, it is easy to see that ACs do not provide CDA-freedom, because ACs will allow $\mathbb{E}_\bot[c_A(s_1)]$ to run to completion. Technically, $\text{CDAF}(\mathbb{E}_\bot, H, \rightarrow_A)$ does not hold for all heaps H.

On the other hand, it can be shown that Cs does prevent this CDA, i.e., $\text{CDAF}(\mathbb{E}_\bot, H, \rightarrow_C)$ holds for all H. The intuition is that if the attacker is able to write any capability ${}^{\mathbb{W}} r$ into r_O, then it must be able to compute ${}^{\mathbb{W}} r$, which implies from the expression evaluation rules of Cs that $\bot \geq_{\mathbb{L}} \mathbb{O}(r)$. Hence, $\bot$ can write to r and, so, clause 2 must hold for r.

Cs do not prevent all CDAs: Based on the above example, one may speculate that Cs prevent all CDAs. However, as the following three examples show, this speculation is false.

Example 2 (Value attack)**.** In this example, we do not allow the adversary to control the location that is written, but instead allow it to control the value that is written. This could, for instance, model a SQL injection attack on a high integrity database, via a confused deputy such as a web server. Assume that $r \in AIS$, $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r)$ and $\bot \geq_{\mathbb{L}} \mathbb{O}(r')$, so $\bot$ cannot write to r directly, but it can write to r'. Consider $\mathbb{E}_\bot = \bot\{\bullet\} \circ \top\{{}^{\mathbb{W}} r := !({}^{\mathbb{R}} r')\}$. This context simply copies the contents of r' into r. This context also does not satisfy $\text{CDAF}(\mathbb{E}_\bot, H, \rightarrow_C)$ for all H. To see this, consider the adversary $c_\bot = ({}^{\mathbb{W}} r' := 42)$ with $H(r) \neq 42$. Then, $\mathbb{E}_\bot[c_\bot]$ ends with 42 in r. Clause 1 does not hold because for $c'_\bot = ({}^{\mathbb{W}} r' := 41)$, $\mathbb{E}_\bot[c'_\bot]$ ends with 41 in r. Clause 2 does not hold because no code running in $\bot\{\bullet\}$ can write 42 (or any value) to r.

Example 3 (Implicit influence)**.** Consider the following context: $\mathbb{E}_\bot = \bot\{\bullet\} \circ \top\{\text{if } (!^{\mathbb{R}} r_A) \text{ then } {}^{\mathbb{W}} r_H := 41 \text{ else } {}^{\mathbb{W}} r_H := 42\}$, where $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r_H)$ and $\bot \geq_{\mathbb{L}} \mathbb{O}(r_A)$. This context writes either 41 or 42 to a reference r_H that the adversary cannot write, depending on a boolean read from a reference r_A that the adversary can write. This context has a CDA

and it does not satisfy CDAF($\mathbb{E}_\perp, H, \rightarrow_C$) for all H. To see this consider any H with $H(r_H) \neq 41$ and the adversary command $c_\perp = ({}^{\mathbb{W}}r_A := tt)$. Then, for the reference r_H, neither clause (1) nor (2) holds.

Example 4 (Initial heap attack)**.** Consider the following context: $\mathbb{E}_\perp = \perp\{\bullet\} \circ \top\{!^{\mathbb{R}}r_A := !^{\mathbb{R}}r'_A\}$ where $\perp \geq_{\mathbb{L}} \mathbb{O}(r_A) = \mathbb{O}(r'_A)$. Assume that the initial heap is such that $H(r_A) = {}^{\mathbb{W}}r_H$ with $\perp \not\geq_{\mathbb{L}} \mathbb{O}(r_H)$, i.e., r_A contains a reference ${}^{\mathbb{W}}r_H$ that the adversary cannot write. This context has a CDA — CDAF($\mathbb{E}_\perp, H, \rightarrow_C$) does not hold for all heaps H. To see this, consider the adversary $c_\perp = ({}^{\mathbb{W}}r'_A := 42)$ and an initial heap H such that $H(r_H) \neq 42$. Then, $\mathbb{E}_\perp[c_\perp]$ ends with 42 in r_H. Clause 1 does not hold because for $c'_\perp = ({}^{\mathbb{W}}r'_A := 41)$, $\mathbb{E}_\perp[c'_\perp]$ ends with 41 in r_H. Clause 2 does not hold because no code running in $\perp\{\bullet\}$ can write 42 to r_H.

Note that there is a fundamental difference in the nature of the CDA in Examples 1, 2, 3 and 4. In Example 1, the deputy (region $\top$) obtains the write capability to the reference under attack from the adversary. We refer to this kind of capability designation as *explicit*. In Examples 2, 3 and 4 the deputy already has the capability (either directly in its code or indirectly through the initial heap) but the adversary influences what gets written to it. We refer to this kind of designation as *implicit*. As should be clear from the examples, Cs prevent CDAs caused by explicit designation (Example 1) but do not prevent CDAs caused by implicit designation.

In Section III-B, we show that the language can be restricted to rule out cases with implicit designation. Trivially, for this restricted language, Cs prevent all CDAs. However, this restricted language also rules out many harmless programs. But before going into that, we point out a shortcoming of our current definition of CDA-freedom and propose a fix.

Relaxing CDA-freedom: Our current definition of CDA-freedom *completely* rules out the possibility that the adversary influence any reference of interest that it cannot write directly. In practice, it is possible that the deputy allows the adversary to have *controlled influence* on a privileged reference. The billing file from the compiler example at the beginning of Section III is a good example. There, the adversary (compiler invoker) can legitimately influence the billing file, e.g., by changing the size of the source file, but the deputy (compiler service) wants to limit this control by allowing only legitimate billing values to be written to the billing file.

To permit such controlled interaction between the adversary and the deputy, we introduce a notion of endorsement (on the lines of information flow endorsement [20]). We allow a region to be declared endorsed (denoted by $\overline{\mathbb{P}}$ as opposed to $\mathbb{P}$), and subsequently be taken out of the purview of the CDA-freedom definition. The intuition is to distinguish, via an endorsed region, a confused deputy from a deputy which is acting on an attacker's authority on purpose. We propose the following definition of CDA-freedom with endorsement (denoted by CDAF-E). CDAF-E essentially states that for an authority context ($\mathbb{E}_{\rho_A}$), heap (H) and a reduction relation ($\rightarrow_{red}$), CDAF should hold for all subsequences of region commands which do not include any endorsed principal.

Definition 3 (CDA-freedom with endorsement)**.** Context $\mathbb{E}_{\rho_A} = \rho_1\{c_1\} \circ \ldots \circ \rho_n\{c_n\}$ is called CDA-free with endorsement under heap H and semantics $\rightarrow_{red}$, written CDAF-E($\mathbb{E}_{\rho_A}, H, \rightarrow_{red}$), if for every contiguous subsequence $\mathbb{E}'_{\rho_A} = \rho_i\{c_k\} \circ \ldots \circ \rho_j\{c_{k+m}\}$ of $\mathbb{E}_{\rho_A}$ such that for all $i \in \{k, \ldots, k+m\}$, ρ_i is not of the form $\overline{\mathbb{P}}$ for any $\mathbb{P}$, we have CDAF($\mathbb{E}'_{\rho_A}, H, \rightarrow_{red}$).

The parameter H in CDAF-E($\mathbb{E}_{\rho_A}, H, \rightarrow_{red}$) represents any heap starting from which we wish to test non-endorsed subsequences of region commands in $\mathbb{E}_{\rho_A}$. It may sound odd that we use the same heap to test all such subsequences, but the intent is to universally quantify over H outside the definition, so specifying a separate starting heap for each subsequence is not useful.

Example 5 (Compiler service)**.** We extend the compiler service (Example 1) with the billing file. Assume that the billing amount for a source file s is computed by the function $\texttt{billing}(s)$ and that the billing file is represented by the reference r_B with $\perp \not\geq_{\mathbb{L}} \mathbb{O}(r_B)$. Then, we can write the complete compiler as the context: $\mathbb{E}_\perp = \perp\{\bullet\} \circ \top\{(!^{\mathbb{R}}r_O) := \texttt{compile}(!^{\mathbb{R}}r_S)\} \circ \overline{\top}\{{}^{\mathbb{W}}r_B := \texttt{billing}(!^{\mathbb{R}}r_S)\}$. Note that this context has the same CDA as the simplified one from Example 1 (the adversary can confuse the compiler by passing a privileged reference in r_O). Correctly, this context does not satisfy Definition 3. However, importantly, it does not fail this definition because of the third region command $\overline{\top}\{{}^{\mathbb{W}}r_B := \texttt{billing}(!^{\mathbb{R}}r_S)\}$, which writes a controlled value derived from an adversary controlled reference r_S to a privileged reference r_B. That region command is endorsed by $\overline{\top}$ and, hence, excluded from the purview of the definition. Instead, the context fails the definition due its first two region commands $\perp\{\bullet\} \circ \top\{(!^{\mathbb{W}}r_O) := \texttt{compile}(!^{\mathbb{R}}r_S)\}$, which indeed have an undesirable CDA.

B. Capability semantics prevent some CDAs

Examples 2, 3 and 4 show that the capability semantics, Cs, cannot prevent all CDAs even in our simple region calculus. In this subsection, we explore this point further and show that under very strong restrictions on programs (contexts) and heaps, Cs do to prevent all CDAs. We introduce some terminology for discourse. Given an attacker region ρ_A, we call a region ρ low integrity or low if $\rho_A \geq_{\mathbb{L}} \rho$. Dually, a region ρ is high integrity or high if $\rho_A \not\geq_{\mathbb{L}} \rho$. A reference r is called low (high) if $\mathbb{O}(r)$ is low (high), i.e., if ρ_A can

(cannot) directly write the reference.

From our examples, it should be clear that if a high region ends up possessing a high reference r in AIS, then Cs alone may not prevent all CDAs because Cs' checks are limited to references only and, hence, an adversary could confuse the high region by influencing values that the high region writes to a high reference in AIS. Consequently, if we wish to use Cs to prevent all CDAs, we must place enough restrictions on contexts to ensure that high regions never end up possessing high references from AIS (low references are not a concern for preventing CDAs because these references always satisfy clause 2 of Definition 2). The converse is also trivially true: If no high region ever possesses a high reference from AIS, then no high reference from AIS can ever be written under Cs semantics, so clause 1 of Definition 2 must hold for all high references in AIS.

There are three ways in which a high region may end up possessing a high reference from AIS. First, the command that starts running in the high region may have a hard-coded high reference from AIS, as in Examples 2 and 3. Second, the command in the high region may read the high reference from another reference, as in Example 4. Third, the adversary may pass the high reference through another reference, as in Examples 1 and 5. Checks made by Cs prevent the adversary from ever evaluating (let alone passing) a high reference, so the third possibility is immediately ruled out in Cs semantics. It follows, then, that if we can restrict our language and heaps substantially to prevent the first two possibilities, then Cs semantics will imply CDA-freedom.

To prevent the first two possibilities, we restrict initial commands in high regions and the initial heap. Accordingly, we create the following two definitions, which say, respectively, that the commands in non-endorsed high regions and the (initial) heap do not have high references from AIS.

Definition 4 (No interesting high references in high regions)**.** A context $\mathbb{E}_{\rho_A}$ has no interesting high references in non-endorsed high regions, written $nihrP(\mathbb{E}_{\rho_A})$, if $\mathbb{E}_{\rho_A} = \rho_1\{c_1\}\circ\ldots\circ\rho_n\{c_n\}$ and for all $i \in \{1,\ldots,n\}$, if $\rho_A \not\geq_{\mathbb{L}} \rho_i$, $\rho_i \neq \overline{\mathbb{P}}$ and ${}^{\mathbb{W}}r \in c_i$, then either $\rho_A \geq_{\mathbb{L}} \mathbb{O}(r)$ or $r \notin AIS$.

Definition 5 (No interesting high references in heap)**.** A heap H has no interesting high references, written $nihrH(H, \rho_A)$ if for all r, $H(r) = {}^{\mathbb{W}}r'$ implies either $\rho_A \geq_{\mathbb{L}} \mathbb{O}(r')$ or $r' \notin AIS$.

We now state the main result of this section: If the initial heap has no interesting high references and the context has no interesting high references in non-endorsed high regions, then the context has no CDAs under Cs.

Theorem 4 (Cs prevents some CDAs)**.** If $nihrH(H, \rho_A)$ and $nihrP(\mathbb{E}_{\rho_A})$, then CDAF-E$(\mathbb{E}_{\rho_A}, H, \rightarrow_C)$.

Proof: We first show that the absence of high references of AIS from the heap and non-endorsed high regions is invariant under $\rightarrow_C$. This implies that no high reference from AIS is ever written in the execution of $\mathbb{E}_{\rho_A}[c_A]$. Hence, clause 1 of Definition 2 holds for all high references in AIS and clause 2 holds for all low references in AIS. ■

We note that the restrictions in the condition of this theorem are extremely strong. In the next section, we present alternative mechanisms for obtaining CDA-freedom that relax these restrictions, at the expense of more runtime overhead. Also note that, if preventing CDAs were the only objective, then Cs are very imprecise: They block many programs that have no CDAs.

Example 6. Consider the context $\bot\{\bullet\} \circ \top\{{}^{\mathbb{W}}r := 1\}$ that simply writes 1 to the reference r. Assume $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r)$. Clearly, this context does not have a CDA as the adversary can control neither the reference that is written (always r) nor the value that is written (always 1), so clause 1 of the CDA-freedom definition holds for all references. However, when instantiated with any adversarial command c_A that computes a high reference, the resulting program will be stopped by Cs.

IV. CDA PREVENTION USING PROVENANCE TRACKING

In this section, we describe two mechanisms other than Cs for preventing CDAs. Both mechanisms relax the assumptions needed for CDA prevention (the pre-conditions of Theorem 4) and, at the same time, execute several CDA-free programs like Example 6, which Cs block. We present the mechanisms as two alternative semantics for our calculus. Both semantics start from the same baseline — the access control semantics (ACs) — and add checks based on *provenance tracking* to prevent CDAs. Provenance tracking, which is based on the extensively studied taint tracking (e.g., [24]), augments ACs to *label* each computed value with a principal, which is a lower bound on the principals whose references have been read to compute the value. Since code in region ρ can only write to references below the principal corresponding to the region ρ, the principal labeling a value is also a lower bound on the principals whose *code* has influenced the value. With such a labeling mechanism in place, CDAs can be prevented easily by checking during reference assignment (rule A-Assign) that an attacker-influenced value is not written to a high reference.

Our two new semantics differ in how they compute labels. Our first semantics, called the explicit-only provenance semantics or EPs, tracks regions that have influenced a value but ignores the effect of implicit influences due to control flow. As a result, this semantics prevents CDAs only under some assumptions, but these assumptions are still weaker than those needed for preventing CDAs via Cs (i.e., the assumptions of Theorem 4). Our second semantics, called the full provenance semantics or FPs, tracks all influences on a value, including implicit ones. This semantics prevents

all CDAs without additional assumptions. Since EPs does not track implicit influences, it can be implemented far more efficiently than FPs (this is well-known from work on information flow control), which justifies our interest in both semantics, not just FPs.

A. Explicit-only provenance semantics

The explicit-only provenance semantics (EPs) tracks, for every computed value, the principals whose references have affected the value. Only explicit influences, such as those due to reference copying are tracked. EPs does not track influence due to control constructs (branch conditions in conditionals and loops). We start from the access control semantics, ACs, and modify the expression evaluation judgment $\langle H, e\rangle \overset{\rho}{\Downarrow_A} v$ to include a label (a principal) on the output value v. This label is a lower bound on all principals whose references have been read during the computation of e. To avoid confusion, we denote labels with the letter ℓ, but readers should note that like $\mathbb{P}$'s, labels are drawn from $Prin$.

The revised judgment for expression evaluation is written $\langle H, e\rangle \overset{\rho}{\Downarrow_{EP}} v^{\ell}$. Its rules are shown in Figure 3. In rule EP-Val, the expression e is already a value. Computation of the result does not read any reference, so the label on the output is $\top$ (the highest point of the lattice $\mathbb{L}$). In rule EP-Deref, the expression being evaluated has the form $!e$. In this case, the semantics first evaluates e to the read capability of a reference ${}^{\mathbb{R}}r$ and then dereferences r. The result could be influenced by every region that was dereferenced in computing r from e as well as $\mathbb{O}(r)$. Hence, the output label is the meet or greatest lower bound ($\sqcap$) of the label of r (denoted ℓ_r in the rule) and $\mathbb{O}(r)$.

The command and program evaluation relations of EPs are written $\overset{\rho}{\rightarrow_{EP}}$ and $\rightarrow_{EP}$, respectively. They use the rules of ACs (Figure 2), except the rule for assignment, which now makes an additional check to ensure that low-influenced values are not written to high references. This revised rule, EP-Assign, is also shown in Figure 3. In comparison to the ACs rule, A-Assign, there is one additional last premise. This premise checks that the label on the updated reference (called ℓ_r) and the label on the value written to the reference (called ℓ_v) are both above (higher integrity than) the principal that owns the reference. This ensures that if the updated reference is high (unwritable by the adversary directly), then the value written has no low (adversarial) influence. Importantly, the check is made only if the executing region is not endorsed.

As in Theorem 4, to show that EPs ensures CDA-freedom, we must assume that the initial commands in high regions do not contain high references from AIS (condition $nihrP(\mathbb{E}_{\rho_A})$). However, the condition on the initial heap in Theorem 4—that the heap contain no high references from AIS—can now be weakened slightly: We only require that *high references* in the initial heap not contain any high references from AIS. Intuitively, we do not care about the contents of the low references in the initial heap because anything read from low references will carry a low label (by rule EP-Deref) and, hence, cannot influence anything written to a high reference (rule EP-Assign).

Definition 6 (No interesting high references in high heap)**.** A heap H has no interesting high references in high parts, written $nihrHH(H, \rho_A)$ if for all r, $\rho_A \not\geq_{\mathbb{L}} \mathbb{O}(r)$ and $H(r) = {}^{\mathbb{W}}r'$ imply either $\rho_A \geq_{\mathbb{L}} \mathbb{O}(r')$ or $r' \notin AIS$.

Note that $nihrH(H, \rho_A)$ immediately implies $nihrHH(H, \rho_A)$ so the latter is a weaker property and hence constitutes a weaker assumption.

Theorem 5 (EPs prevents some more CDAs)**.** If $nihrHH(H, \rho_A)$ and $nihrP(\mathbb{E}_{\rho_A})$, then CDAF-E($\mathbb{E}_{\rho_A}, H, \rightarrow_{EP}$).

Proof: We first show that the absence of high references of AIS from the high part of the heap and non-endorsed high regions is invariant under $\rightarrow_{EP}$. This implies that no high reference from AIS is ever written in the execution of $\mathbb{E}_{\rho_A}[c_A]$. Hence, clause 1 of Definition 2 holds for all high references in AIS and clause 2 holds for all low references in AIS. ■

We now discuss the relative permissiveness of Cs and EPs for CDA-prevention.

Example 7. This examples highlights the difference between the assumptions $nihrHH(H, \rho_A)$ of Theorem 5 and $nihrH(H, \rho_A)$ of Theorem 4. Consider the context $\top\{{}^{\mathbb{W}}r := !({}^{\mathbb{R}}r_L)\}$, which doesn't even contain a hole for the adversary (and, hence, trivially, has no CDA). Consider the adversary level $\rho_A = \bot$, assume that $\bot \geq_{\mathbb{L}} \mathbb{O}(r_L) = \mathbb{O}(r)$ and that we start from a heap H with $H(r_L) = {}^{\mathbb{W}}r_H$ with $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r_H)$. Then, it is easy to see that $nihrHH(H, \rho_A)$, so the assumption of Theorem 5 does not rule this program out, but it is not the case that $nihrH(H, \rho_A)$, so the assumption of Theorem 4 does rule this program out.

We saw earlier that Example 6 has no CDA, but is halted by Cs. It can be easily checked that EPs allows the example to execute to completion. Based on this, one may ask whether EPs is strictly more permissive than Cs when the program passes the conditions of Theorem 4 (and, hence, also of Theorem 5). However, this is false as the following example shows.

Example 8. Consider the context $\rho\{{}^{\mathbb{W}}r_H := !^{\mathbb{R}}r_L\}$, which has no hole for the adversary and, hence, no CDA. Assume that $\rho_A = \rho$ and $\rho \geq_{\mathbb{L}} \mathbb{O}(r_H) > \mathbb{O}(r_L)$. Then, since the context copies a value from a reference r_L to r_H and the owner of the former is strictly below the owner of the latter, EPs will stop this context from executing. On the other hand, Cs will allow this context to execute to completion.

$$\text{EP-Val}\ \frac{}{\langle H, v\rangle \overset{\rho}{\Downarrow}_{EP} v^{\top}} \qquad \text{EP-Deref}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{A} {}^{\mathbb{R}}r^{\ell_r} \qquad \mathbb{P}_r = \mathbb{O}(r) \qquad v = H(r)}{\langle H, !e\rangle \overset{\rho}{\Downarrow}_{EP} v^{\ell_r \sqcap \mathbb{P}_r}}$$

$$\text{EP-Assign}\ \frac{\langle H, e_1\rangle \overset{\rho}{\Downarrow}_{EP} {}^{\mathbb{W}}r^{\ell_r} \qquad \rho \geq_{\mathbb{L}} \mathbb{O}(r) \qquad \langle H, e_2\rangle \overset{\rho}{\Downarrow}_{EP} v^{\ell_v} \qquad \rho \neq \overline{\mathbb{P}} \implies \ell_r \sqcap \ell_v \geq_{\mathbb{L}} \mathbb{O}(r)}{\langle H, e_1 := e_2\rangle \rightarrow^{\rho}_{EP} \langle H[r \mapsto v], \text{skip}\rangle}$$

Figure 3. Explicit provenance semantics (all other rules are same as those of access control semantics, Figure 2)

Hence, EPs prevents all CDAs on a slightly larger language fragment than Cs (Theorem 4 vs Theorem 5). However, the permissiveness of EPs and Cs on CDA-free programs in the *common* fragment is incomparable (Examples 6 and 8).

B. Full provenance semantics

We now show that by tracking complete provenance of values, including implicit influences due to control flow, we can enforce CDA-freedom for our entire calculus (without any restrictions on contexts or heaps). To do this, we build a full provenance semantics or FPs for our calculus. FPs is based upon similar semantics for information flow control [24], with minor adjustments to account for regions. As in EPs, every computed value is labeled with a principal, which is a lower bound on principals whose references (and code) could have influenced the value. To track influence due to control flow, we introduce an auxiliary label to the semantic state. This label, called the program counter or pc in information flow control literature, is a lower bound on all regions that have influenced the reachability of the current command. When we enter the body of a control construct like if-then-else or while, we lower the pc to the meet of the current pc and the label of the branch or loop condition. When we exit the body of the control construct we restore the pc back to its original value (not restoring the pc would make the semantics less permissive). To enable this restoration, we maintain a stack of pc's and push the new pc to the stack when we enter an if-then-else or while construct. We pop the stack when we exit the construct. This stack is denoted PC. Its topmost label is the current pc. At the top-level, we start with $PC = [\top]$.

The rules for FPs are shown in Figure 4. The expression evaluation judgment is identical to that in EPs; it has the form $\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell}$. The judgment for reducing commands is now modified to include the stack PC. It takes the form $\langle H, PC, c\rangle \rightarrow^{\rho}_{FP} \langle H', PC', c'\rangle$. When entering the body of an if-then-else or while construct (rules FP-if, FP-else and FP-while 1), we push $pc \sqcap \ell$ onto PC, where pc is the current topmost label on PC and ℓ is the label of the branch or loop condition. We also add a marker (endif or endwhile) to the code body to indicate when the body ends. When this marker is encountered, we pop the stack PC (rules FP-endif and FP-endwhile). By doing this, we ensure that the top label on PC is a lower bound on the labels of all branch/loop conditions that influence the control flow at the current instruction. The most interesting rule of the semantics is that for assignment (rule FP-Assign), which, in addition to all checks made by the corresponding rule in EPs, also checks that the current pc is above the owning region of the reference being written (last premise). This additional check ensures that an adversary cannot influence the contents of high references even through control constructs, as in Example 3.

The judgment for evaluating programs, $\langle H, PC, P\rangle \rightarrow_{FP} \langle H', PC', P'\rangle$, also carries PC but its rules do not modify PC in any interesting way. It is an invariant that $PC = [\top]$ at the beginning of the program's execution and every time a region's command ends with skip. This explains why $PC = [\top]$ in rule FP-Comp 2.

Next, we show that FPs enforces CDA-freedom without any assumptions. Technically, the definitions of CDA-freedom, Definition 2 and Definition 3, do not directly apply to FPs because those definitions assume that the reduction semantics $\rightarrow_{red}$ rewrite pairs $\langle H, P\rangle$, whereas the FPs reduction $\rightarrow_{FP}$ rewrites triples $\langle H, PC, P\rangle$. However, we can reinterpret $\langle H, P\rangle$ in Definition 2 to mean $\langle H, [\top], P\rangle$. With that implicit change, we can prove the following theorem.

Theorem 6 (FPs prevent all CDAs)**.** CDAF-E($\mathbb{E}_{\rho_A}, H, \rightarrow_{FP}$) holds unconditionally.

Proof: We prove that FPs enforces a strong form of information flow integrity (Definition 8), which in turn implies CDA-freedom with endorsement (Theorem 7). To prove the former, we build a simulation relation between states of the form $\langle H, PC, c\rangle$, as is standard in information flow control [24]. ■

We now discuss the relative permissiveness of FPs and EPs for CDA-prevention.

Example 9. This example demonstrates a CDA-free context that violates the assumptions of Theorem 5 and, hence, is not in the fragment on which EPs enforces CDA-freedom. Consider Example 7, but stipulate that $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r_L)$. The heap now violates $nihrHH(H, \rho_A)$, hence, the resulting example lies outside the fragment allowed by the assumptions of Theorem 5. However, the example will execute to

Expressions:

$$\text{FP-Val}\ \frac{}{\langle H, v\rangle \overset{\rho}{\Downarrow}_{FP} v^{\top}} \qquad \text{FP-Deref}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} {}^{\mathbb{R}}r^{\ell_r} \qquad \mathbb{P}_r = \mathbb{O}(r) \qquad v = H(r)}{\langle H, !e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell_r \sqcap \mathbb{P}_r}}$$

Commands:

$$\text{FP-if}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell} \qquad v = tt}{\langle H, pc :: PC, \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \rightarrow^{\rho}_{FP} \langle H, (pc \sqcap \ell) :: pc :: PC, c_1; \text{endif}\rangle}$$

$$\text{FP-else}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell} \qquad v = ff}{\langle H, pc :: PC, \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \rightarrow^{\rho}_{FP} \langle H, (pc \sqcap \ell) :: pc :: PC, c_2; \text{endif}\rangle}$$

$$\text{FP-while 1}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell} \qquad v = tt}{\langle H, pc :: PC, \text{while } e \text{ do } c\rangle \rightarrow^{\rho}_{FP} \langle H, (pc \sqcap \ell) :: pc :: PC, c; \text{endwhile; while } e \text{ do } c\rangle}$$

$$\text{FP-while 2}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v \qquad v = ff}{\langle H, PC, \text{while } e \text{ do } c\rangle \rightarrow^{\rho}_{FP} \langle H, PC, \text{skip}\rangle} \qquad \text{FP-endwhile}\ \frac{}{\langle H, pc :: PC, \text{endwhile}\rangle \rightarrow^{\rho}_{FP} \langle H, PC, \text{skip}\rangle}$$

$$\text{FP-Assign}\ \frac{\langle H, e_1\rangle \overset{\rho}{\Downarrow}_{FP} {}^{\mathbb{W}}r^{\ell_r} \qquad \langle H, e_2\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell_v} \qquad \rho \geq_{\mathbb{L}} \mathbb{O}(r) \qquad \rho \neq \overline{\mathbb{P}} \implies \ell_r \sqcap \ell_v \sqcap pc \geq_{\mathbb{L}} \mathbb{O}(r)}{\langle H, pc :: PC, e_1 := e_2\rangle \rightarrow^{\rho}_{FP} \langle H[r \mapsto v], pc :: PC, \text{skip}\rangle} \qquad \text{FP-endif}\ \frac{}{\langle H, pc :: PC, \text{endif}\rangle \rightarrow^{\rho}_{FP} \langle H, PC, \text{skip}\rangle}$$

$$\text{FP-Seq 1}\ \frac{\langle H, PC, c_1\rangle \rightarrow^{\rho}_{FP} \langle H', PC' c_1'\rangle}{\langle H, PC, c_1; c_2\rangle \rightarrow^{\rho}_{FP} \langle H', PC', c_1'; c_2\rangle} \qquad \text{FP-Seq 2}\ \frac{}{\langle H, PC, \text{skip}; c_2\rangle \rightarrow^{\rho}_{FP} \langle H, PC, c_2\rangle}$$

Program:

$$\text{FP-Prg 1}\ \frac{\langle H, PC, c\rangle \rightarrow^{\rho}_{FP} \langle H', PC', c'\rangle}{\langle H, PC, \rho\{c\}\rangle \rightarrow_{FP} \langle H', PC', \rho\{c'\}\rangle} \qquad \text{FP-Comp 1}\ \frac{\langle H, PC, P_1\rangle \rightarrow_{FP} \langle H', PC', P_1'\rangle}{\langle H, PC, P_1 \circ P_2\rangle \rightarrow_{FP} \langle H', PC', P_1' \circ P_2\rangle}$$

$$\text{FP-Comp 2}\ \frac{}{\langle H, [\top], \rho\{\text{skip}\} \circ P\rangle \rightarrow_{FP} \langle H, [\top], P\rangle}$$

Figure 4. Full provenance semantics

completion in FPs.

On contexts that lie in the fragment allowed by EPs, FPs is no more permissive than EPs. This is easy to see: FPs makes additional checks for implicit influences, which EPs does not. In fact, due to these checks, FPs is strictly less permissive on the EPs fragment, as the following example shows.

Example 10. Consider the context $\rho\{\text{if } (!^{\mathbb{R}}r_A) \text{ then } {}^{\mathbb{W}}r_H := 41 \text{ else } {}^{\mathbb{W}}r_H := 42\}$ with $\rho_A = \rho \geq_{\mathbb{L}} \mathbb{O}(r_H) > \mathbb{O}(r_A)$. This code has no CDA since it has no hole for the adversary. FPs will stop this program because a location of lower integrity (r_A) influences a location of higher integrity (r_H) via control flow. However, EPs will allow this program to execute to completion because it does not track such influences.

Summary of examples and CDA-prevention: To summarize, the three semantics Cs, EPs and FPs soundly enforce CDA-freedom with endorsement for progressively larger sublanguages, with FPs covering our entire language. However, for programs and heaps that satisfy the assumptions of Cs, EPs and Cs are incomparable in permissiveness. On the subset of programs and heaps that satisfy the assumptions of EPs, EPs is strictly more permissive than FPs. The access control semantics, ACs, is ineffective against confused deputy attacks, even those with only explicit designation. The following table summarizes how all our examples fare on all four semantics. A and R mean that the semantics would accept and reject (halt) the program respectively. For programs with CDA, we write A or R to mean accept or reject when the adversary tries to launch a CDA. NP

means that the example is outside the fragment on which the semantics enforces CDA-freedom.

Example	Has CDA?	ACs	Cs	EPs	FPs
1	CDA	A	R	R	R
2	CDA	A	NP	R	R
3	CDA	A	NP	NP	R
4	CDA	A	NP	R	R
5	CDA	A	R	R	R
6	no CDA	A	NP	NP	A
7	no CDA	A	NP	A	A
8	no CDA	A	A	R	R
9	no CDA	A	NP	NP	A
10	no CDA	A	A	A	R

V. Relation to information flow integrity

Our formalization of CDA-freedom, Definitions 2 and 3, is inspired by the notion of information flow integrity. Here, we compare the two. A standard baseline definition for information flow integrity is Goguen and Meseguer (GM) style non-interference [25], which states that a program satisfies integrity if the high parts of the final memory obtained by executing the program cannot be influenced by the low parts of the initial memory. Since Definition 2 talks about a program with holes, we modify GM-style non-interference to take holes into account. This yields Definitions 7 and 8 (these definitions are inspired by [22]). For an attacker ρ_A, say that H_1 and H_2 are high-equivalent, written $H_1 \sim_{\rho_A} H_2$, if for all r such that $\rho_A \not\geq_{\mathbb{L}} \mathbb{O}(r)$, it is the case that $H_1(r) = H_2(r)$.

Definition 7 (Non-interference without endorsement)**.** A context $\mathbb{E}_{\rho_A}$ has non-interference against active adversaries under reduction semantics $\rightarrow_{red}$, written *NI-A*$(\mathbb{E}_{\rho_A}, \rightarrow_{red})$, if for all heaps H_1, H_2, H_1', H_2' and commands c_A, c_A', if $H_1 \sim_{\rho_A} H_2$, $\langle H_1, \mathbb{E}_{\rho_A}[c_A]\rangle \rightarrow^*_{red} \langle H_1', \texttt{final}\rangle$ and $\langle H_2, \mathbb{E}_{\rho_A}[c_A']\rangle \rightarrow^*_{red} \langle H_2', \texttt{final}\rangle$, then $H_1' \sim_{\rho_A} H_2'$.

Definition 8 (Non-interference with endorsement)**.** Context $\mathbb{E}_{\rho_A} = \rho_1\{c_1\} \circ \ldots \circ \rho_n\{c_n\}$ has non-interference against active adversaries with endorsement under reduction semantics $\rightarrow_{red}$, written *NI-A-E*$(\mathbb{E}_{\rho_A}, \rightarrow_{red})$, if for every contiguous subsequence $\mathbb{E}'_{\rho_A} = \rho_i\{c_k\} \circ \ldots \circ \rho_j\{c_{k+m}\}$ of $\mathbb{E}_{\rho_A}$ such that for all $i \in \{k, \ldots, k+m\}$, ρ_i is not of the form $\overline{\mathbb{P}}$ for any $\mathbb{P}$, we have *NI-A*$(\mathbb{E}'_{\rho_A}, \rightarrow_{red})$.

It turns out that Definition 7 is strictly stronger than Definition 2 (and consequently Definition 8 is strictly stronger than Definition 3). Intuitively, Definition 7 ensures that all high references in AIS satisfy clause 1 of Definition 2 (all low references trivially satisfy clause 2 in all four of our semantics).

Theorem 7 (Non-interference implies CDA-freedom)**.** For $\rightarrow_{red} \in \{\rightarrow_A, \rightarrow_C, \rightarrow_{EP}, \rightarrow_{FP}\}$, *NI-A-E*$(\mathbb{E}_{\rho_A}, \rightarrow_{red})$ implies CDAF-E$(\mathbb{E}_{\rho_A}, H, \rightarrow_{red})$ for every H.

The converse of this theorem does not hold, as the following counterexample shows. The reason is that the definition of non-interference quantifies over *two* initial heaps H_1 and H_2, while the definition of CDA-freedom does not.

Example 11. Consider the context $\mathbb{E}_{\rho_A} = \rho_H\{^{\mathbb{W}}r_H := \ !^{\mathbb{R}}r_A\}$, where $\rho_A \not\geq_{\mathbb{L}} \mathbb{O}(r_H)$ and $\rho_A \geq_{\mathbb{L}} \mathbb{O}(r_A)$. The context copies the contents of r_A to r_H, so this context does not satisfy Definition 8. Specifically, we can choose any H_1 and H_2 that agree on all references except r_A. Then, $H_1 \sim_{\rho_A} H_2$. However, the final heaps disagree on r_H, so the final heaps are not equivalent. On the other hand, this context trivially satisfies Definition 3 since it has no hole for the adversary.

VI. Related Work

Understanding precisely which security properties a capability system can enforce and how capabilities and access control differ are long-standing research topics. We focus our discussion only on work dealing with these two issues.

Using informal arguments, Miller *et al.* [11] compare four system models for taking authorization decisions: the first two models are access control as columns and capabilities as rows of the Lampson matrix [6]. The third model is capabilities as keys (as in the Amoeba operating system [4]) and the fourth model is capabilities as objects (as in the Joe-E language [26]). Our capability semantics (Cs) can be seen as an abstraction of either model 2 or a degenerate case of model 4, while our access control semantics (ACs) are an abstraction of model 1. Miller *et al.*'s comparison is based on 7 properties A...G, where only property A is claimed to be impossible to hold in access control as columns. (Although properties B...G are, in practice, not implemented in access control, there is no reason in theory not to have an access control system complying with B...G). Starting from this comparison, we decided to focus our formalization of capabilities only taking property A into account. Miller *et al.* discuss three "myths": the equivalence myth, the confinement or delegation myth, and the irrevocability myth. In this paper, we only consider the equivalence myth.

The confinement myth deals with "capabilities cannot limit the propagation of authority". Miller *et al.* also discuss CDAs in relation to capabilities. CDAs, first described in the literature in 1988 [16], are related to property A — *No Designation without Authority* — since authorization given to one party T (the deputy) is used to access a resource designated by a different party U (the adversary).

The topmost implication of Fig. 14 in [11] and other works such as [16], [23] seem to suggest that CDAs are impossible in capability systems with Property A. Our contribution is to clarify the arguments of [11] and include examples of CDAs that can happen with Property A and those that cannot. Indeed, in some cases of CDAs, designation of a resource can be done *implicitly* by U, in contrast

to explicit designation which would clearly be prevented by Property A.

Chander *et al.* [27] use state-transition systems to model capabilities and access control. They model two versions of capabilities, based on [1]. The first model is capabilities as rows in the Lampson matrix and the second model is capabilities as unforgeable tokens. None of these models have property A. Hence, their capabilities are more similar to our access control model than to our capabilities model. In particular, in their second capabilities model where a capability to access a resource r is an unforgeable token T(r), there is nothing that prevents a subject s from designating r even though s cannot generate a capability T(r). Moreover, once s possesses T(r), s can pass this capability to other subjects. In contrast, our capability semantics would prevent this. With the goal of comparing both systems according to their power of delegation, Chander *et al.* prove various simulation relations between capabilities and access control. One of their main results is the equivalence between access control and capabilities viewed as rows of the Lampson access matrix (without modeling property A). In contrast, we show that the equivalence does not hold with property A. Chander *et al.* also prove that there is no equivalence between access control and capabilities when properties of capabilities seen as unforgeable tokens are taken into account. In their model, this is due to the impossibility of revoking capabilities. We have not modeled revocation in our calculus since it is not needed for exploring the equivalence myth.

Maffeis *et al.* [12] formally connect capabilities that are objects [2] to operational semantics of programming languages. Capability safety refers to the property of a language that guarantees that a component must have a capability to access a resource. They explicitly formalize property A (see §V, Def. 8, cond. 1(b) of [12]) as a basic condition of an object capability system. They do not directly explain how property A can be modeled in an operational semantics, which is central to our results. They also prove that Cajita, a component of Caja [3]—an object capability language based on JavaScript—is capability safe.

Murray *et al.* [28] define an object capability model in the CSP process algebra. In their model, they do not formalize property A and, in particular, they allow for delegation of capabilities as in [27]. These points differ from our model. Using a model checker, Murray *et al.* can detect covert channels of illegal information flows [21]. Their work focuses on the detection of information flow leaks. Our CDA-freedom is similar to, but slightly weaker than, information flow integrity [22] and our work is focused on prevention, not detection. Our full provenance analysis ensures information flow integrity (and thus CDA-freedom).

Reasoning about the correct use of capabilities, separating security policies from implementations, has been studied in [29] and [30]. Drossopoulou and Noble [29] analyze Miller's Mint and Purse example of [2] (in capabilities based on JAVA as in, e.g., Joe-E [26]) using a formal specification language. Building on object capabilities, Drossopoulou *et al.* [31] propose special specification predicates in a specification language based on JavaScript and simpler than the one of [29]. With these specification predicates they can model risk and trust in systems having components with different levels of trust. In their specification language, they cannot directly express the idea of encapsulation of objects. We conjecture that explicitly allowing encapsulation of objects to appear in specifications should be tantamount to assuming property A in object capability systems. Along the same lines, Saghafi *et al.* [23] informally discuss a relation between Property D of [11] and encapsulation in order to compare capabilities with feature-oriented programming. Dimoulas *et al.* [30] propose extensions to capability languages that restrict the propagation of capabilities according to declarative policies. By means of integrity policies, they restrict components that may influence the use of a capability. They do not model property A.

Birgisson *et al.* [32] propose secure information flow enforcement by means of capabilities. They do so by proposing a transformation from arbitrary source programs to a language with capabilities. (In their experiments, the target language is Caja [3].) They present formal guarantees of information flow security [21] and permissiveness.

VII. Conclusion

We examine the relation between access control and capability semantics in a simple language setting. We model Miller *et al.*'s property A "no designation without authority", but without appealing to object encapsulation as in object capability languages [2]. We also present the first extensional characterization of freedom from confused deputy attacks (CDAs) and relate it to information flow integrity. We clarify which classes of CDAs capabilities (as a mechanism) can and cannot prevent and stipulate the exact conditions under which they can prevent all classes of CDAs. Furthermore, we present alternate ways of preventing CDAs using provenance tracking with fewer conditions.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG) grant "Information Flow Control for Browser Clients" under the priority program "Reliably Secure Software Systems" (RS^3) and the ANR project AJACS ANR-14-CE28-0008.

References

[1] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984.

[2] M. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, Johns Hopkins University, 2006.

[3] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe active content in sanitized javascript." [Online]. Available: http://code.google.com/p/google-caja

[4] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, and S. J. Mullender, "Experiences with the amoeba distributed operating system," *Comm. ACM*, 1990.

[5] Mozilla Developer Network, "Script security." [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Script_security

[6] B. Lampson, "Protection," *Operating Systems Review*, vol. 8, no. 1, pp. 18–24, Jan. 1974.

[7] R. Sandhu, "Role-based access control," *Advances in Computers*, vol. 46, pp. 237–286, 1998.

[8] Ú. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *IEEE Symposium on Security and Privacy (Oakland)*, 2000.

[9] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode, "Enforcing authorization policies using transactional memory introspection," in *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[10] D. Garg and F. Pfenning, "Noninference in constructive authorization logic," in *IEEE Computer Security Foundations Workshop (CSFW)*, 2006.

[11] M. S. Miller, K.-P. Yee, J. Shapiro *et al.*, "Capability myths demolished," Johns Hopkins University Systems Research Laboratory, Tech. Rep. SRL2003-02, 2003. [Online]. Available: http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf

[12] S. Maffeis, J. C. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[13] S. Drossopoulou and J. Noble, "The need for capability policies," in *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2013.

[14] L. Jia, S. Sen, D. Garg, and A. Datta, "A logic of programs with interface-confined code," in *IEEE Computer Security Foundations Symposium (CSF)*, 2015.

[15] D. Devriese, L. Birkedal, and F. Piessens, "Reasoning about object capabilities with logical relations and effect parametricity," in *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[16] N. Hardy, "The confused deputy (or why capabilities might have been invented)," *Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[17] OWASP, "Cross site request forgery." [Online]. Available: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29

[18] CERT, "Ftp bounce attacks." [Online]. Available: http://www.cert.org/historical/advisories/CA-97.27.FTP_bounce.cfm

[19] L. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *USENIX Security Symposium*, 2012.

[20] P. Li, Y. Mao, and S. Zdancewic, "Information integrity policies," in *Workshop on Formal Aspects in Security and Trust (FAST)*, 2003.

[21] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[22] C. Fournet and T. Rezk, "Cryptographically sound implementations for typed information-flow security," in *ACM Symposium on Principles of Programming Languages (POPL)*, 2008.

[23] S. Saghafi, K. Fisler, and S. Krishnamurthi, "Features and object capabilities: Reconciling two visions of modularity," in *International Conference on Aspect-oriented Software Development (AOSD)*, 2012.

[24] G. Boudol, "Secure information flow as a safety property," in *Workshop on Formal Aspects in Security and Trust (FAST)*, 2008.

[25] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy (Oakland)*, 1982.

[26] A. Mettler, D. Wagner, and T. Close, "Joe-E: A security-oriented subset of Java," in *Network and Distributed System Security Symposium (NDSS)*, 2010.

[27] A. Chander, J. C. Mitchell, and D. Dean, "A state-transition model of trust management and access control," in *IEEE Computer Security Foundations Workshop (CSFW)*, 2001.

[28] T. C. Murray and G. Lowe, "Analysing the information flow properties of object-capability patterns," in *Formal Aspects in Security and Trust (FAST)*, 2009.

[29] S. Drossopoulou and J. Noble, "How to break the bank: Semantics of capability policies," in *International Conference on Integrated Formal Methods (iFM)*, 2014.

[30] C. Dimoulas, S. Moore, A. Askarov, and S. Chong, "Declarative policies for capability control," in *IEEE Computer Security Foundations Symposium (CSF)*, 2014.

[31] S. Drossopoulou, J. Noble, and M. S. Miller, "Swapsies on the Internet: First steps towards reasoning about risk and trust in an open world," in *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2015.

[32] A. Birgisson, A. Russo, and A. Sabelfeld, "Capabilities for information flow," in *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2011.