Authority Analysis for Least Privilege Environments

Toby Murray and Gavin Lowe

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom
{toby.murray, gavin.lowe}@comlab.ox.ac.uk

Abstract. The rise of limited-privilege environments has been accompanied by the emergence of vulnerabilities in which a subject is able to maliciously wield their limited privileges to indirectly cause unwanted effects. Unfortunately, conventional safety analyses for access control systems are ill-equipped to deal with this problem because they do not detect the indirect effects that a subject can cause, but merely the permissions a subject can acquire.
We present a technique that characterises a subject's authority as all of the effects they can cause to occur. Our technique is based on an analysis of causation, applied to a CSP model of a system. These analyses can be expressed as CSP refinements and, hence, automatically performed by a refinement-checker such as FDR. We demonstrate the ability of our technique to successfully identify excess authority by examining the "Confused Deputy" scenario, whose vulnerability goes undetected with conventional safety analyses.

Key words: Access control, authority, causation, least privilege, formal definition, automatic verification, CSP, model checking, security.

1 Introduction

1.1 The Rise of Least Privilege

There appears to be a growing consensus that a failure to adhere to the principle of *least privilege* [25] has led current mainstream computing systems to be far less secure than they might have been. Executable email attachments are dangerous because, when opened by a user on current conventional operating systems, they are allowed to cause any effect that the user herself is allowed to cause. The severity of a remote code-execution vulnerability in a network-facing application is proportional to the privileges granted to that application.

Recently, many different solutions have been proposed and implemented in an attempt to rectify this situation. Some [19, 32, 12, 27, 11, 35] have provided startlingly clear examples of what the future might hold for secure computing if current applications and systems were reimplemented on top of architectures that naturally support the principle of least privilege. Other attempts [33, 3, 15,

20, 31, 26] have shown the challenges involved in trying to retrofit least privilege to the current mainstream computing base.

A common feature of all of these systems is the means to confine the set of *permissions* available to a running instance of an application. By permissions, we mean the set of objects in the system that the instance can access, or interact with, directly. In order to adhere to the principle of least privilege, instances are initially given minimal sets of permissions. While an instance is running it may acquire new permissions as its function alters. For example, when a user opens a new document in a word processor, the word processor might be granted the permission to read the file that contains the document. Indeed, in order to adhere to the principle of least privilege, an instance *must* be able to acquire new privileges as it is running. Otherwise, instances must be given the union of all permissions they might ever need, completely violating the principle of least privilege.

However, this leads to a potential problem. If instances can acquire new permissions, how can one be sure that a running instance cannot acquire a permission that it should not be allowed to have? For example, how can one be sure that a word processor won't be able to obtain the permission to write to the kernel binary?

This is an instance of the *safety problem* [7] for access control systems, which seeks to determine whether a particular subject can ever acquire a particular permission. Many formal models and decision procedures have been proposed [7, 1, 30, 9] in order to reason about this problem. When designing and analysing a limited-privilege system, it is imperative to be able to apply safety analyses in order to show that a running instance cannot obtain extra privileges in excess of the minimum required to perform its function at the current time.

1.2 Re-Enter An Old Attack

Unfortunately, as limited-privilege solutions are becoming more widely available, we are beginning to see classes of attack emerging that had previously been confined to the academic community [6]. These are attacks in which one subject s is able to use their permission to access another subject t to cause t to perform some action on s's behalf that violates the principle of least privilege.

The prototypical example [6] of this attack carries the name "Confused Deputy". In this attack, one subject, Alice, has access to another subject, Carol, a compiler. Alice has permission to invoke Carol with an output filename. Carol has permission to write to a special purpose billing file, Bill, in which Carol maintains a log of her own usage. By invoking Carol with the name of Bill, Alice can indirectly cause Bill to be overwritten, despite the fact that Alice does not have permission to write to Bill. Here Carol's permission is being used incorrectly on behalf of Alice.

A real-world example of this scenario is described by Spiessens [28]. Here, a user of a dynamic-firewall application grants network access to all instances of the ssh program, in order to allow her to connect securely to remote machines without having to click through a firewall dialogue each time. Unfortunately,

in doing so, she has turned ssh into the ultimate confused deputy. Any other application that can execute ssh can now gain indirect access to an encrypted, authenticated channel to any remote host on the Internet. Among other things, this provides a useful path of egress for any piece of spyware on the system. Here ssh's permission is being used incorrectly on behalf of a malicious piece of spyware.

Another example [34] of this attack involves the User Account Control [33] (UAC) feature of Windows Vista. Under UAC, when an ordinary application tries to perform a sensitive function that requires administrator privileges, UAC displays a dialogue that may allow the user to grant the privileges to the application. Applications that are part of the operating system produce a dialogue labelled "Windows needs your permission to continue". The executable RunLegacyCPLElevated allows legacy dynamic libraries to be identified by UAC as part of the operating system. RunLegacyCPLElevated is invoked with a dynamic library filename as its argument, which it then executes on the invoker's behalf, thereby allowing the executing code to be identified as part of Windows. We argue that RunLegacyCPLElevated is a confused deputy. A subject, Dave, with the permission to execute RunLegacyCPLElevated and to write dynamic libraries to disk now has the indirect ability to cause arbitrary code to be executed that will be identified by UAC as part of Windows. Here RunLegacyCPLElevated's permission is being used incorrectly on behalf of Dave.

Current models for the analysis of the safety problem are ill-equipped to reason about the indirect effects that a subject can cause by use of its permissions. We refer to all such indirect effects as a subject's *authority*. Traditional safety analyses are limited to reasoning about authority in terms of the direct permissions a subject can acquire. As shown by the examples above, this can grossly underestimate a subject's total authority. As we shall demonstrate later, in the first example a simple safety analysis fails to reveal that Alice has authority to overwrite Bill, since Alice never has permission to overwrite Bill. Similarly, a simple safety analysis would fail to reveal the excess authority in the second and third examples as well.

These vulnerabilities highlight the importance of the principle of *least authority* [18, 17]. It is not enough to simply limit a subject's permissions in order to enforce meaningful least privilege. As we shall see later on, in the Confused Deputy scenario, limiting Alice's permissions to the smallest reasonable set still provides Alice with excess authority. In order to provide meaningful security, we require methods that can correctly analyse and detect excess authority once a subject's permissions have been minimised.

1.3 Contribution

In this paper, we present a technique specifically designed to reason about the indirect effects that a subject may be able to cause, in order to give an upper bound on the subject's authority. We use the process algebra CSP [21], and its stable-failures denotational semantics, to model and reason about causation and thereby define those actions that a subject may indirectly cause to occur.

CSP has been successfully applied to reasoning about many security-relevant systems, problems and properties including the safety problem in access control systems [10, 2], information flow [16], cryptographic protocols [23] and real-world security policies [22]. We begin with a brief overview of the syntax and semantics of CSP in Section 2.

In Section 3 we define authority in terms of all the events an object can cause to occur. Excess authority, then, is authority not allowed by the security policy. We give a straightforward definition of causation, which we call *Traces-Causation*. Unfortunately, this definition suffers from the refinement paradox: there are processes in which a particular event e cannot be caused by some object o, but that have refinements in which o can cause e to occur. We argue that this is undesirable, and that we should consider a definition of causation that holds whenever a process has a refinement for which Traces-Causation holds. We then give an alternative, equivalent, characterisation of causation, in terms of the traces and failures of the process.

In Section 4 we show how this property can be tested for using a model checker such as FDR [14]. In Section 5 we model the Confused Deputy scenario; we show how a simple safety analysis fails to detect the attack, but that our technique accurately detects Alice's authority to cause Bill to be overwritten. In Section 6 we sum up, compare and contrast our technique to related work, and consider some areas for future work.

2 A brief overview of CSP

In this section we give a brief overview of CSP. More details can be obtained elsewhere [21].

CSP is a process algebra, with various semantics, for describing and reasoning about concurrent systems. A system modelled in CSP comprises a set of concurrently executing *processes*. Each process usually models some particular component of the system in question. Processes execute by performing *events*. An event represents an atomic communication; this might either be between two processes or between a process and the environment. Processes communicate with each other by synchronising on common events. We write Σ for the set of all visible events.

2.1 Syntax

The process $STOP$ can perform no events. The process $a \rightarrow P$ can perform the event a, and then act like P. The process $?a : A \rightarrow P_a$ offers the set of events A; if a particular event a is performed, the process then acts like P_a. (The prefixing operator "$\rightarrow$" binds tighter than all other operators.)

CSP allows multi-part events where each part is separated by an infix dot ".", such as the event $\mathsf{up}.3$. The process $\mathsf{up}?a : A \rightarrow P_a$ initially offers the set of events $\{\mathsf{up}.a \mid a \in A\}$. The output operator "!" is used to offer specific events to the environment. The process $\mathsf{move}?x{:}X!3 \rightarrow P$ initially offers the set

of events $\{\mathsf{move}.x.3 \mid x \in X\}$. The notation $\{|\mathsf{move}|\}$ denotes the set of events whose first part is move. The first part of a multi-part event is sometimes called a *channel*.

The process $P \Box Q$ represents an external choice between P and Q; the initial events of both processes are offered to the environment; when an event is performed, that resolves the choice. $P \sqcap Q$ represents an internal or nondeterministic choice between P and Q; the process can act like either P or Q, with the choice being made according to some criteria that we do not model. The process $P \lhd b \rhd Q$ acts like P if the boolean condition b is true; otherwise it acts like Q.

The process $CHAOS_A$ is the most nondeterministic, nondivergent process with alphabet A; it can perform any sequence of events from A, and refuse any events.

If c is a channel, then $c.P$ represents the process that acts like P except every event x is renamed to $c.x$.

$P \underset{A}{\parallel} Q$ represents the parallel composition of P and Q, synchronising on events from A. For a set of processes, $\{P_1, \ldots, P_n\}$, and a set of alphabets $\{A_1, \ldots, A_n\}$ (each a subset of Σ), $\Big\|_{1 \le i \le n} (P_i, A_i)$ represents the parallel composition of the P_i, where each P_i is allowed to perform events only from the set A_i, and all processes that share a common event must synchronise on it. $P \,|||\, Q$ represents the interleaving of P and Q, i.e., parallel composition with no synchronisation.

2.2 Semantics

A *trace* is a sequence of visible events that a process can perform. We write $traces(P)$ for the set of all traces of P.

We write traces within angle brackets ($\langle \ldots \rangle$). $s \mathbin{\hat{}} t$ denotes the concatenation of s and t. $s \upharpoonright A$ denotes the trace obtained by taking s and removing all events not in A. $s \setminus A$ denotes the opposite: the trace obtained by taking s and removing all events in A: $s \setminus A = s \upharpoonright (\Sigma - A)$. Trace s is said to be a *prefix* of t, written $s \le t$, if there exists a sequence u, where $t = s \mathbin{\hat{}} u$. Within traces, the special event $\surd$ represents termination, and can occur only at the end of a trace.

A *stable failure* is a pair (s, X), where s is some trace that P can perform, and X is a set of events that P can stably refuse after performing s: i.e., after s, P can reach a state where no internal activity is possible and none of the events from X is possible. We write $failures(P)$ for the set of all stable failures of P.

A *divergence* is a trace after which a process can diverge, i.e., perform an infinite amount of internal activity. We write $divergences(P)$ for the set of divergences of P. All of the processes we consider in this paper are *divergence free*. In this case the traces and stable failures are related by:

$$traces(P) = \{s \mid (s, X) \in failures(P)\}.$$

Within the failures-divergences semantics of CSP, a process is represented by its failures and its divergences. The following axioms hold for the failures F and divergences D of a process P.

F1. $traces(P) = \{t \mid (t, X) \in F\}$ is non-empty and prefix closed.
F2. $(v, X) \in F \wedge Y \subseteq X \Rightarrow (v, Y) \in F$.
F3. $(v, X) \in F \wedge v^\frown\langle a\rangle \notin traces(P) \Rightarrow (v, X \cup \{a\}) \in F$.
F4. $v^\frown\langle\surd\rangle \in traces(P) \Rightarrow (v, \Sigma) \in F$.
D1. $s \in D \cap \Sigma^* \wedge t \in \Sigma^{*\surd} \Rightarrow s^\frown t \in D$.
D2. $s \in D \Rightarrow (s, X) \in F$.
D3. $s^\frown\langle\surd\rangle \in D \Rightarrow s \in D$.

We say that process Q refines process P, written $P \sqsubseteq Q$, when

$$failures(Q) \subseteq failures(P) \wedge divergences(Q) \subseteq divergences(P).$$

3 Characterising Authority

In this section, we produce a semantic characterisation of causation.

The systems we model comprise a set of *objects*. (By object, we mean both subjects and objects as defined in the standard access control literature: we make no distinction between the two.) Each object has a set of events that constitute its *alphabet*. These are the events that the object is able to partake in. Intuitively, if some object can perform some events from its alphabet that can, perhaps indirectly, cause some other event e to occur, then that object has authority to cause e.

We assume that the system is modelled by a CSP process. For simplicity, we restrict ourselves to divergence-free processes in this paper (i.e. those that can never perform an infinite amount of internal activity without communicating with their environment), and assume a finite alphabet.

3.1 Defining Causation

We base our understanding of causation on Lewis' counterfactual definition [13] that states that x causes y if y would be possible if x had occurred, but y would not be possible if x had not occurred. This leads to a simple definition for causation that can be applied to the traces of a process P. An object o with alphabet A can cause some event e to occur if there is some trace s after which e can follow, but when the events of A are removed from s, e cannot follow. We define the predicate $TC_P(A, e)$ (Traces-Causation) to capture this.

$$TC_P(A, e) \mathrel{\widehat{=}} \exists\, s \bullet s^\frown\langle e\rangle \in traces(P) \wedge (s \setminus A)^\frown\langle e\rangle \notin traces(P).$$

Often, we will be interested in the negation of this predicate, i.e. when the event e cannot be caused by any object with alphabet A. Thus, we define the predicate

$$NTC_P(A, e) \mathrel{\widehat{=}} \neg TC_P(A, e).$$

Unfortunately, this definition is too strong when applied to nondeterministic processes. Consider the process

$$P = a \rightarrow b \rightarrow STOP \sqcap b \rightarrow STOP,$$

and suppose the nondeterminism is resolved to the left. In this case, a can certainly cause b to occur; however, P doesn't satisfy the above definition of causation since both $\langle a, b\rangle$ and $\langle b\rangle$ are in $traces(P)$.

The problem here is that P is refined by processes, such as $Q = a \rightarrow b \rightarrow STOP$, in which a can cause b to occur. The refinements of P represent all of the possible ways in which nondeterminism in P can be resolved. As argued in [16], it is important to be able to detect when some property does not hold for a refinement of P, despite the fact that it does hold for P itself. P will often represent the design of a system, rather than its implementation; in this case, nondeterminism in P might represent underspecification which is to be resolved when the system is implemented. At other times, P might represent a model of a system in which nondeterminism is used to abstract away from low-level details of the real system in question. In both cases, it's important to be sure that, however the nondeterminism is resolved, the original property will still hold.

This realisation leads to the predicate $C_P(A, e)$ that holds precisely when P has a refinement for which $TC_P(A, e)$ holds:

$$C_P(A, e) \mathrel{\widehat{=}} \exists\, Q \bullet P \sqsubseteq Q \wedge TC_Q(A, e).$$

The predicate

$$NC_P(A, e) \mathrel{\widehat{=}} \neg C_P(A, e)$$

captures the negation of $C_P(A, e)$. Thus, NC is the *refinement-closure* of NTC:

$$NC_P(A, e) \equiv \forall\, Q \bullet P \sqsubseteq Q \Rightarrow NTC_Q(A, e).$$

The above property appears difficult to test, because of the quantification over all refinements of P. In order to derive a method for testing if a process satisfies NC, we give an alternative characterisation of causation. Here, an event e can be caused by an object o with alphabet A if there exists some trace of the system, in which o participates, after which e can follow; but when the events of A are removed, it's possible that e cannot follow, in the sense that e or an earlier event can be refused. We define the predicate $FC_P(A, e)$ (Failures-Causation) as follows:

$$\begin{aligned} FC_P(A, e) \mathrel{\widehat{=}} \exists\, s, t \bullet\ & s \mathbin{\hat{}} t \mathbin{\hat{}} \langle e\rangle \in traces(P) \wedge s \upharpoonright A \neq \langle\rangle \wedge \\ & (s \setminus A, \{first(t \setminus A \mathbin{\hat{}} \langle e\rangle)\}) \in failures(P). \end{aligned}$$

Notice that the process P defined above does satisfy this definition of causation. We can see that $FC_P(\{a\}, b)$ holds by taking $s = \langle a\rangle$ and $t = \langle\rangle$, since $\langle a, b\rangle \in traces(P)$ and $(\langle\rangle, \{b\}) \in failures(P)$.

We will show, below, that $C_P(A, e) \equiv FC_P(A, e)$. We begin by showing that, under FC, non-causation is refinement-closed.

Lemma 1. *If* $\neg FC_P(A, e)$ *and* $Q \sqsupseteq P$ *then* $\neg FC_Q(A, e)$.

Proof. Suppose, for a contradiction, that $\neg FC_P(A, e)$, $Q \sqsupseteq P$ and $FC_Q(A, e)$. Then for some s, t:

$$s \mathbin{\hat{}} t \mathbin{\hat{}} \langle e\rangle \in traces(Q) \wedge s \upharpoonright A \neq \langle\rangle \wedge (s \setminus A, \{first(t \setminus A \mathbin{\hat{}} \langle e\rangle)\}) \in failures(Q).$$

But $traces(Q) \subseteq traces(P) \wedge failures(Q) \subseteq failures(P)$ so

$$s \hat{} t \hat{} \langle e \rangle \in traces(P) \wedge s \upharpoonright A \neq \langle\rangle \wedge (s \setminus A, \{first(t \setminus A \hat{} \langle e \rangle)\}) \in failures(P),$$

contradicting $\neg FC_P(A, e)$. □

We now show that for divergence-free processes, Traces-Causation implies Failures-Causation.

Lemma 2. *If Q is non-divergent and $TC_Q(A, e)$ then $FC_Q(A, e)$.*

Proof. Consider a divergence-free process Q for which $TC_Q(A, e)$ holds. Then there is some trace s such that

$$s \hat{} \langle e \rangle \in traces(Q) \wedge (s \setminus A) \hat{} \langle e \rangle \notin traces(Q).$$

Thus $s \neq s \setminus A$, so $s \upharpoonright A \neq \langle\rangle$. Partition s about its first event a from A, into the sequence $t \hat{} \langle a \rangle \hat{} u$ so

$$s = t \hat{} \langle a \rangle \hat{} u \wedge a \in A \wedge t \upharpoonright A = \langle\rangle.$$

Then

$$s \hat{} \langle e \rangle = t \hat{} \langle a \rangle \hat{} u \hat{} \langle e \rangle \in traces(Q) \wedge (s \setminus A) \hat{} \langle e \rangle = t \hat{} (u \setminus A) \hat{} \langle e \rangle \notin traces(Q).$$

Now, $t \in traces(Q)$, so let v be the longest prefix of u such that $t \hat{} (v \setminus A) \in traces(Q)$, and let w be the remainder of u:

$$v \hat{} w = u \wedge t \hat{} (v \setminus A) \in traces(Q) \wedge t \hat{} (v \setminus A) \hat{} \langle first(w \setminus A \hat{} \langle e \rangle) \rangle \notin traces(Q).$$

Now Q is divergence-free, so $(t \hat{} (v \setminus A), \{\}) \in failures(Q)$, and hence by Axiom **F3** (see Section 2),

$$(t \hat{} (v \setminus A), \{first(w \setminus A \hat{} \langle e \rangle)\}) \in failures(Q).$$

Combining the above results we have

$$\begin{array}{l} t \hat{} \langle a \rangle \hat{} v \hat{} w \hat{} \langle e \rangle \in traces(Q) \wedge \\ ((t \hat{} \langle a \rangle \hat{} v) \setminus A, \{first(w \setminus A \hat{} \langle e \rangle)\}) \in failures(Q), \end{array}$$

and hence $FC_Q(A, e)$ holds. □

We now use these lemmas to prove that Causation and Failures-Causation are equivalent:

Theorem 1. *For any divergence-free process, P:*

$$C_P(A, e) \equiv FC_P(A, e).$$

Proof. We begin by showing that $C_P(A,e) \Rightarrow FC_P(A,e)$. So suppose that $C_P(A,e)$, i.e., that there exists $Q \sqsupseteq P$ such that $TC_Q(A,e)$. Then Q must also be divergence-free, so by Lemma 2, $FC_Q(A,e)$ holds. Hence, by Lemma 1, $FC_P(A,e)$ holds.

Conversely, suppose $FC_P(A,e)$ holds. Then for some s and t:

$$s \mathbin{\hat{}} t \mathbin{\hat{}} \langle e \rangle \in traces(P) \wedge s \upharpoonright A \neq \langle\rangle \wedge (s \setminus A, \{c\}) \in failures(P),$$
where $c = first(t \setminus A \mathbin{\hat{}} \langle e \rangle)$.

We construct a process Q that refines P and such that $TC_Q(A,e)$. Define Q to be the divergence-free process that has the same failures as P, except with all those corresponding to the trace $s \setminus A \mathbin{\hat{}} \langle c \rangle$ removed:

$$\begin{aligned} failures(Q) = failures(P) - \\ \{(s \setminus A \mathbin{\hat{}} \langle c \rangle \mathbin{\hat{}} t, X) \mid t \in \Sigma^*, X \subseteq \Sigma\} - \\ \{(s \setminus A, X) \mid (s \setminus A, X \cup \{c\}) \notin failures(P)\}. \end{aligned}$$

Lemma 3, which follows, shows that such a process exists.

Observe that $P \sqsubseteq Q$ since $failures(Q) \subseteq failures(P)$.

Now, Q is divergence-free, so $traces(Q) = \{v \mid (v,X) \in failures(Q)\}$. Hence

$$s \mathbin{\hat{}} t \mathbin{\hat{}} \langle e \rangle \in traces(Q),$$

since this is not one of the traces removed. Also, $s \setminus A \mathbin{\hat{}} \langle c \rangle \leq (s \setminus A) \mathbin{\hat{}} (t \setminus A) \mathbin{\hat{}} \langle e \rangle = (s \mathbin{\hat{}} t) \setminus A \mathbin{\hat{}} \langle e \rangle$ so

$$(s \mathbin{\hat{}} t) \setminus A \mathbin{\hat{}} \langle e \rangle \notin traces(Q).$$

Hence, $TC_Q(A,e)$ holds. □

Finally, we must show that the process Q constructed in Theorem 1 exists. We will need the following result from [21, Section 9.3]:

Theorem 2. *Assuming the alphabet Σ is finite, for any choice of (F,D) that satisfies the axioms (see Section 2) of the failures-divergences model of CSP, there is a CSP process Q whose failures and divergences are F and D respectively.*

Hence it will be enough to show that $failures(Q)$ and $divergences(Q)$ satisfy the axioms.

Lemma 3. *$failures(Q)$, as defined in Theorem 1, and $divergences(Q) = \{\}$ satisfy the axioms* **F1**–**F4** *and* **D1**–**D3** *of the failures-divergences model.*

Proof. Observe that since $divergences(Q) = \{\}$, Axioms **D1**–**D3** hold trivially. We consider each of the remaining axioms in turn. Note that

$$traces(Q) = traces(P) - \{s \setminus A \mathbin{\hat{}} \langle c \rangle \mathbin{\hat{}} t \mid t \in \Sigma^*\}.$$

Axiom F1. Clearly $traces(Q)$ is non-empty: it contains, at least, the empty trace. It is prefix-closed since $traces(P)$ is, and we remove an extensions-closed set of traces.

Axiom F2. Q satisfies **F2** since P does, and whenever we remove a failure, we remove all failures with larger refusal sets.

Axiom F3. Suppose $(v, X) \in failures(Q)$ and $v \,\hat{}\, \langle a \rangle \notin traces(Q)$. Then $(v, X) \in failures(P)$. We perform a case analysis.

- Case $v \,\hat{}\, \langle a \rangle \neq s \setminus A \,\hat{}\, \langle c \rangle$. Then $v \,\hat{}\, \langle a \rangle \notin traces(P)$, and so $(v, X \cup \{a\}) \in failures(P)$, since P satisfies **F3**. And hence $(v, X \cup \{a\}) \in failures(Q)$, by construction.
- Case $v = s \setminus A \wedge a = c$. Recall that $(s \setminus A, \{c\}) \in failures(P)$. Hence $(s \setminus A, X \cup \{c\}) \in failures(P)$, by Axiom **F2**. And hence $(v, X \cup \{a\}) = (s \setminus A, X \cup \{c\}) \in failures(Q)$, by construction.

Axiom F4. Suppose $v \,\hat{}\, \langle \surd \rangle \in traces(Q)$. Then $v \,\hat{}\, \langle \surd \rangle \in traces(P)$ and $s \setminus A \,\hat{}\, \langle c \rangle \not\leq v$. Then $(v, \Sigma) \in failures(P)$ since P satisfies **F4**. Hence $(v, \Sigma) \in failures(Q)$, by construction, whether or not v equals $s \setminus A$. □

4 Testing for Authority

We now construct a refinement test, which can be automatically carried out by a model checker such as FDR [14], that checks whether $NC_P(A, e)$ holds, i.e., that checks that an object with alphabet A cannot cause e to occur. Note that we restrict ourselves to finite-state processes, where this question is decidable. We generalise the test from a single event e to a set of events B, where $A \cap B = \{\}$: we define $NC_P(A, B) \mathrel{\widehat{=}} \forall\, e \in B \bullet NC_P(A, e)$.

The test works as follows. We run two copies of P in parallel, in a harness, with a controller (or scheduler). Initially, we allow only the left-hand copy of P to perform A events, and force both copies to do the same non-A events. At some point, after the left-hand copy has done at least one event from A, and has just performed some event $c \notin A$, we pause the right-hand copy of P. At this point, the left-hand copy will have performed some trace $s \,\hat{}\, \langle c \rangle$ with $s \upharpoonright A \neq \langle\rangle$ and the right-hand copy will have performed $s \setminus A$. Following the definition of FC, we continue to run the left-hand copy until it has performed a trace $s \,\hat{}\, t \,\hat{}\, \langle e \rangle$, for some $e \in B$ and trace t; since $c \notin A$ we will have $c = first(t \setminus A \,\hat{}\, \langle e \rangle)$. At this point we restart the right-hand copy of P and test whether it can refuse the event c; if so, $FC_P(A, B)$ holds.

The events of the left- and right-hand copies of P are distinguished by using a renaming transformation that has each copy perform its events on separate fresh channels, left and right. The harness in which the two copies of P are run with the controller is defined as follows.

$$Harness(P) = (\mathsf{left}.P \;|||\; \mathsf{right}.P) \underset{\{|\mathsf{left},\mathsf{right}|\}}{\|} Ctrl1.$$

Here, $\mathsf{left}.P$ ($\mathsf{right}.P$) denotes the process that performs the event $\mathsf{left}.x$ ($\mathsf{right}.x$) whenever P performs x. The controller process, $Ctrl1$, is defined as follows.

$$\begin{aligned}
Ctrl1 &= \mathsf{left}?c \rightarrow (Ctrl2 \,{}_{\not}\!\!\blacktriangleleft c \in A \blacktriangleright \mathsf{right}.c \rightarrow Ctrl1),\\
Ctrl2 &= \mathsf{left}?c \rightarrow (Ctrl2 \,{}_{\not}\!\!\blacktriangleleft c \in A \blacktriangleright (\mathsf{right}.c \rightarrow Ctrl2 \sqcap \mathsf{ping} \rightarrow Ctrl3(c))),\\
Ctrl3(c) &= Ctrl5(c) \,{}_{\not}\!\!\blacktriangleleft c \in B \blacktriangleright Ctrl4(c),\\
Ctrl4(c) &= \mathsf{left}?d \rightarrow (Ctrl5(c) \,{}_{\not}\!\!\blacktriangleleft d \in B \blacktriangleright Ctrl4(c)),\\
Ctrl5(c) &= \mathsf{ping} \rightarrow \mathsf{right}.c \rightarrow STOP.
\end{aligned}$$

The controller initially forces the right-hand copy of P to perform the same events as the left-hand copy, until the latter performs an event from A. The controller then (in state $Ctrl2$) continues to force the right-hand copy to perform the same non-A events as the left-hand copy, except after a non-A event c it can (nondeterministically) choose to pause the right-hand copy, signalled by the event ping. It then continues to run the left-hand copy until it performs an event from B; if c itself is in B, then this is immediate (states $Ctrl3$ and $Ctrl4$). The right-hand copy is then re-awoken (in state $Ctrl5$), also signalled by the event ping, in order to test whether it can refuse c.

$Spec1$ is the most general process that mirrors the behaviour of the harness, except that it never refuses the final event $\mathsf{right}.c$.

$$\begin{aligned}
Spec1 &= \mathsf{left}?c \rightarrow (Spec2 \,{}_{\not}\!\!\blacktriangleleft c \in A \blacktriangleright (\mathsf{right}.c \rightarrow Spec1 \sqcap STOP)) \sqcap STOP,\\
Spec2 &= \mathsf{left}?c \rightarrow\\
&\qquad (Spec2 \,{}_{\not}\!\!\blacktriangleleft c \in A \blacktriangleright (\mathsf{right}.c \rightarrow Spec2 \sqcap \mathsf{ping} \rightarrow Spec3(c) \sqcap STOP))\\
&\quad \sqcap STOP,\\
Spec3(c) &= Spec5(c) \,{}_{\not}\!\!\blacktriangleleft c \in B \blacktriangleright Spec4(c),\\
Spec4(c) &= \mathsf{left}?d \rightarrow (Spec5(c) \,{}_{\not}\!\!\blacktriangleleft d \in B \blacktriangleright Spec4(c)) \sqcap STOP,\\
Spec5(c) &= \mathsf{ping} \rightarrow \mathsf{right}.c \rightarrow STOP.
\end{aligned}$$

The states of the specification correspond to the states of the controller. Notice that $Spec5$ cannot refuse to perform $\mathsf{right}.c$. Thus, $Harness(P)$ will refine $Spec1$ if and only if the right-hand copy of P can never refuse the final c event, i.e., if and only if $NC_P(A, B)$ holds. Thus

$$Spec1 \sqsubseteq Harness(P) \;\equiv\; NC_P(A, B).$$

The refinement can be tested using a model checker like FDR. If the refinement fails, FDR will produce a counter-example; the ping events in the counter-example mark the points at which the right-hand copy of P was paused and restarted, and hence aid in its interpretation.

If P has N states then the size of $Harness(P)$ is $O(N^2)$, since it runs two copies of P. In most cases, however, the size of $Harness(P)$ should be significantly less than $O(N^2)$. This is because for each state of the first copy of P, there is likely to be a fairly small number of states that the second copy of P can be in at the same time. If this number is bounded by some constant k, then the total number of states is $O(k.N)$. Hence, in most cases, the time to perform the test should grow linearly with the size of P.

5 Analysing The Confused Deputy

Having described and explained our technique, we now demonstrate its utility for reasoning about Alice's excess authority in the Confused Deputy scenario described in Section 1. First, we model the system in CSP. We then show how a simple safety analysis fails to detect Alice's authority to overwrite Bill, before demonstrating how to accurately detect Alice's excess authority using a refinement check of the sort described in Section 4.

5.1 Modelling the Scenario in CSP

We define a set of operations $Op = \{\mathsf{Read}, \mathsf{Write}, \mathsf{Append}, \mathsf{Exec}\}$ and a set of objects $Object = \{\mathsf{Alice}, \mathsf{Bill}, \mathsf{Carol}\}$[1]. We then define events of the form $o_1.o_2.op$ to represent object o_1 performing operation op on object o_2. The events associated with an Exec operation also carry an object name, and are of the form $o_1.o_2.\mathsf{Exec}.arg$, representing object o_1 executing object o_2, passing the argument arg. Thus we use alphabet

$$\{o_1.o_2.op \mid o_1, o_2 \in Object \wedge op \in \{\mathsf{Read}, \mathsf{Write}, \mathsf{Append}\}\} \cup$$
$$\{o_1.o_2.\mathsf{Exec}.arg \mid o_1, o_2 \in Object \wedge arg \in Object\}.$$

An object o is involved in events that represent it operating on some other object p, and events that represent p operating on it. Hence, the alphabet of each $o \in Object$ is defined as:

$$\alpha(o) = \{|o.p \mid p \in Object - \{o\}|\} \cup \{|p.o \mid p \in Object - \{o\}|\}.$$

Notice that the definition of α is such that an operation is only defined between two distinct objects.

The configuration of permissions is defined by the acl function, which takes an object and an operation and returns the set of objects who have permission to perform that operation on that object: Carol has permission to write and append to Bill; Alice has permission to execute Carol.

$$acl(\mathsf{Bill}, \mathsf{Write}) = \{\mathsf{Carol}\}, \qquad acl(\mathsf{Bill}, \mathsf{Append}) = \{\mathsf{Carol}\},$$
$$acl(\mathsf{Carol}, \mathsf{Exec}) = \{\mathsf{Alice}\}, \qquad acl(other, other) = \{\}.$$

We define a set of parameterised CSP processes that represent the behaviour of different types of entities within the system.

The $Compiler$ process defines the behaviour of a compiler, with identity me, that is able to be executed by any object with Exec permission as defined by its access control list. Once invoked, it writes to the specified file, before appending to its billing file, $logFile$.

$$Compiler(me, logFile) = ?s : acl(me, \mathsf{Exec})!me!\mathsf{Exec}?file \rightarrow$$
$$me.file.\mathsf{Write} \rightarrow$$
$$me.logFile.\mathsf{Append} \rightarrow Compiler(me, logFile).$$

[1] Recall that we use the term "object" to include what are termed "subjects" in some of the access control literature.

The $File$ process defines the behaviour of a file, with identity me, that can be written, appended or read by anyone with the appropriate permission.

$$\begin{aligned} File(me) = {} & ?s : acl(me, \mathsf{Write})!me!\mathsf{Write} \rightarrow File(me) \ \Box \\ & ?s : acl(me, \mathsf{Append})!me!\mathsf{Append} \rightarrow File(me) \ \Box \\ & ?s : acl(me, \mathsf{Read})!me!\mathsf{Read} \rightarrow File(me). \end{aligned}$$

The $User$ process defines the behaviour of a user, with identity me, who tries to execute any program they can, and to read, write and append to any file they can. In this manner, we capture the most general behaviour of any user within our model.

$$\begin{aligned} User(me) = {} & me?prog!\mathsf{Exec}?arg \rightarrow User(me) \ \Box \\ & me?file!\mathsf{Read} \rightarrow User(me) \ \Box \\ & me?file!\mathsf{Write} \rightarrow User(me) \ \Box \\ & me?file!\mathsf{Append} \rightarrow User(me). \end{aligned}$$

The total system, $System$, is then the parallel composition of $User(\mathsf{Alice})$, $File(\mathsf{Bill})$ and $Compiler(\mathsf{Carol}, \mathsf{Bill})$, with the above alphabets.

5.2 A Simple Safety Analysis

We can perform a simple safety analysis [7] to determine whether Alice can ever obtain permission to overwrite Bill. This example uses static access control lists, so clearly it is impossible for Alice to obtain permission to overwrite Bill. However, in more complex examples, the configuration of permissions can change over time, and so a safety analysis of the sort presented here is necessary to determine whether a particular subject can ever acquire a particular permission.

In our example, were Alice able to obtain any permission to Bill, then $System$ would be able to perform some event in $\{|\mathsf{Alice.Bill}|\}$. We can test whether $System$ is ever able to perform such an event by testing if it refines the most general process that performs no such event:

$$Spec = CHAOS_{\Sigma - \{|\mathsf{Alice.Bill}|\}}.$$

FDR indicates that $Spec \sqsubseteq_T System$. As we expect, this simple safety analysis reveals that Alice can never gain permission to overwrite Bill.

5.3 An Authority Analysis

We now analyse whether Alice has authority to cause Bill to be overwritten, i.e., if she can cause some event from $B = \{o.\mathsf{Bill.Write} \mid o \in Object\}$. We can check that Alice has no such authority by testing the following refinement (with $A = \alpha(\mathsf{Alice})$):

$$Spec1 \sqsubseteq Harness(System)$$

FDR completes the test in under a second and indicates that this refinement does not hold. It provides the failure

$$(\langle \mathsf{left.Alice.Carol.Exec.Bill}, \mathsf{left.Carol.Bill.Write}, \mathsf{ping}, \mathsf{ping} \rangle, \{\mathsf{right.Carol.Bill.Write}\})$$

as a counter-example. As expected, the refinement-check reveals that by invoking Carol with the name Bill, Alice can cause Bill to be overwritten (in the left-hand copy), since Carol can refuse to write to Bill if not so invoked (in the right-hand copy).

6 Discussion

The rise of least privilege environments necessitates techniques for formal analysis that can accurately reason about a subject's authority, beyond the set of permissions they can acquire. In particular, the emergence of instances of the Confused Deputy vulnerability demonstrates the need to be able to detect a subject's excess authority in spite of their minimal privileges. Safety analyses are ill-equipped for this task because they are limited to characterising authority in terms of acquirable permissions. As shown, this can grossly underestimate a subject's total authority.

We have presented a technique based on an analysis of causation for reasoning about authority in the presence of least-privilege. We have demonstrated its utility for detecting a subject's excess authority in the Confused Deputy scenario. We hope that as least privilege environments become pervasive, that such analyses will become as important as safety and information-flow analyses are today, in order to ensure that the principle of least authority is upheld.

In the remainder of this section we discuss some related work and prospects for extending the work of this paper.

6.1 Analysing Authority in Capability Systems

In the Confused Deputy example, it is interesting to consider how Alice might be prevented from having authority to overwrite Bill. One solution might be for Carol to check the filename she is passed when invoked and to not write to this file if it is Bill. However this raises the question: what if the subject that executes Carol has permission to write to Bill? Should Carol then write to Bill on that subject's behalf? Unfortunately, as noted by Spiessens [28], Carol does not have the information available to make this determination. Even if she can examine the access control list for Bill, it's possible that Alice is executing her on behalf of some other subject who does have the relevant permission.

When the Confused Deputy was first described [6], it was noted that this problem largely disappears when considered within the context of an access control system that unifies designation and permission, such as those based on capabilities [4]. If Alice can designate Bill to Carol if and only if she herself has permission to Bill, then Carol will write to Bill if and only if Alice has permission to write to Bill, since Carol uses Alice's designation when attempting to write the output. Hence, another possible remedy would be to abandon the use of identity-based access controls, such as the access control lists modelled in the example, in favour of a capability-based approach.

Capability systems are interesting not only because they elegantly solve the Confused Deputy problem. They also naturally support the construction of systems that adhere to the principle of least authority. Further, many of the common abstractions used in capability systems are designed to provide one object with authority to access another, but not direct permission. Hence, being able to reason about authority is crucial for an understanding of many of these abstractions. For these reasons, capability systems present an attractive target for the application of our techniques.

6.2 Pseudo-Permissions in Safety Analyses

Previous attempts to model and reason about authority include efforts based on models for safety-analysis in which *pseudo-permissions* are used to model authority. One such case is the use of *de-facto rights* in Take-Grant systems [1]. For example, a subject s with read permission to subject t, where t is writable by subject r, has *de-facto* read permission to subject r. Unfortunately, the use of de-facto rights fails to take into account the influence that t's behaviour has on s's authority [28]. For example, t might behave in such a way as to prevent any information flowing from r to s, in which case s has no authority to read r. Thus, the use of de-facto rights necessarily over-estimates a subject's total authority. More generally, using pseudo-permissions to model aspects of authority requires specific knowledge about how various permissions may interact with one another in order to give rise to authority.

In contrast, our technique does not rely on any specific knowledge of the interaction between permissions, but rather extracts the causal relationships between events from the semantics of a process. Our approach also allows the restrictive behaviour of a subject to be explicitly modelled via appropriate CSP process definitions. For example, by altering the definition of the *Compiler* process, we could redefine Carol's behaviour in the Confused Deputy example to not write to the file designated by Alice if that file is Bill. An authority analysis would reveal that Carol's restricted behaviour reduces Alice's authority, preventing her from being able to cause Carol to overwrite Bill.

6.3 Non-Interference as the Absence of Authority

The property of *non-interference* [5] and related notions of information-flow can be viewed as characterising the absence of any causal flow from High subjects to Low subjects [24]. We believe that there is a connection between the absence of causal flow and the absence of authority, as defined in this paper. Intuitively, when applying our technique, one subject, High, has no authority over another, Low, if High can never cause some event $l \in \alpha(\mathsf{Low})$ to occur. However, a lack of information flow requires not only that High is unable to cause any event in $\alpha(\mathsf{Low})$ from occurring, but also that High is unable to cause any event in $\alpha(\mathsf{Low})$ from *not* occurring.

We intend to investigate further the relationship between our work and that on information flow; in particular, there seem to be strong similarities with the work of [16].

6.4 Knowledge-Behaviour Models

The work of Spiessens *et al.* [28] on Knowledge-Behaviour Models (KBMs) and the SCOLL language seeks to reason about authority by explicitly taking into account the possible restrictive effects that a subject's behaviour can have on another's authority. This work directly inspired our work on using causal analyses to characterise authority.

Here a subject's permissions and behaviours are represented by a set of predicates, with rules that define how new predicates can be derived from the current set. SCOLL has been applied to reason about indirect authority; however, many examples [28, 8, 29] measure authority in terms of acquirable permissions. As noted in Section 1 and demonstrated by the analysis of the Confused Deputy example in Section 5, this can greatly underestimate a subject's total authority.

Despite this limitation, KBMs have been used to accurately model the Confused Deputy scenario [28]. In order to model the scenario, a behaviour predicate *useForClient* is introduced to describe Carol's intent to use some designation on Alice's behalf. Testing whether *useForClient(*Bob*)* is derivable accurately detects whether Carol is able to use the designation Bob on Alice's behalf.

In this sense, the *useForClient* predicate can be viewed as a means to characterise part of Alice's authority, independently of the permissions Alice can acquire. Unfortunately, this approach requires the incorporation of predicates, like *useForClient*, that explicitly capture the notion of on whose behalf a subject might be acting, in order to reason about authority. In contrast, our approach requires no extra work to be undertaken, since it automatically determines on whose behalf a subject might be acting.

References

1. Matt Bishop and Lawrence Snyder. The transfer of information and authority in a protection system. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 45–54. ACM Press, 1979.
2. Jeremy Bryans. Reasoning about XACML policies using CSP. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 28–35. ACM Press, 2005.
3. Bruno Castro da Silva and Raul Fernando Weber. TuxGuardian: Um firrewall de host voltado para o usuário final. In *5 Fórum Internacional de Software Livre*, 2004. Available at: `http://tuxguardian.sourceforge.net/tg-sbrc.pdf`.
4. Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–154, March 1966.
5. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy 1982*, pages 11–20, 1982.
6. Norm Hardy. The confused deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–38, October 1988.

7. Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 1976.
8. Yves Jaradin, Fred Spiessens, and Peter Van Roy. SCOLL: A language for safe capability based collaboration. Technical Report Research Report INFO-2005-10, Université catholique de Louvain, 2005.
9. Eldar Kleiner and Tom Newcomb. On the decidability of the safety problem for access control policies. In *Sixth International Workshop on Automatic Verification of Critical Systems (AVoCS 2006)*, pages 91–103, 2006.
10. Eldar Kleiner and Tom Newcomb. Using CSP to decide safety problems for access control policies. Technical Report Research Report RR-06-04, Oxford University Computing Laboratory, University of Oxford, January 2006.
11. Ivan Kristić. System security on the One Laptop per Child's XO laptop: The Bitfrost security platform, 2007. Available at: `http://wiki.laptop.org/go/Bitfrost`.
12. Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, June 2005.
13. David Lewis. Causation. *Journal of Philosophy*, 70(17):556–567, 1973.
14. Formal Systems (Europe) Limited. Failures divergences refinement: FDR2 user manual, 2005. Available at: `http://www.fsel.com/documentation/fdr2/fdr2manual.ps`.
15. Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, 2001.
16. Gavin Lowe. On information flow and refinement-closure. In *Proceedings of the Workshop on Issues in the Theory of Security (WITS '07)*, 2007.
17. Mark S. Miller and Jonathan S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Proceedings of the 8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, December 2003.
18. Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. The structure of authority: Why security is not a separable concern. In *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Revised Selected and Invited Papers, LNCS 3389*, pages 2–20. Springer, 2005.
19. Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
20. David S. Peterson, Matt Bishop, and Raju Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
21. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, NJ, USA, 1997.
22. Peter Ryan and Ragni Ryvold Arnesen. A process algebraic approach to security policies. In *DBSec*, pages 301–312, 2002.
23. Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols: the CSP Approach*. Addison Wesley, 2000.
24. Peter Y. A. Ryan. Mathematical models of computer security. In R. Gorrieri, editor, *Proceedings of the 2000 FOSAD Summer School, LNCS 2171*. Springer, 2000.

25. Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1208–1308, September 1975.
26. Mark Seaborn. Plash: tools for practical least privilege, 2007. Available at: `http://plash.beasts.org`.
27. Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *SOSP '99: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, 1999.
28. Alfred Spiessens. *Patterns of Safe Collaboration*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, February 2007.
29. Fred Spiessens, Yves Jaradin, and Peter Van Roy. SCOLL and SCOLLAR: Safe collaboration based on partial trust. Technical Report Research Report INFO-2005-12, Université catholique de Louvain, 2005.
30. Fred Spiessens and Peter Van Roy. A practical formal model for safety analysis in capability-based systems. In *Trustworthy Global Computing, International Symposium, TGC 2005, Revised Selected Papers, LNCS 3705*, pages 248–278. Springer, 2005.
31. Marc Stiegler, Alan H. Karp, Ka-Ping Yee, Tyler Close, and Mark S. Miller. Polaris: Virus safe computing for Windows XP. *Communications of the ACM*, 49(9):83–88, September 2006. Available at: `http://www.hpl.hp.com/techreports/2004/HPL-2004-221.html`.
32. Marc Stiegler and Mark S. Miller. A capability based client: The DarpaBrowser. Technical Report Focused Research Topic 5 / BAA-00-06- SNK, Combex, Inc., June 2002. Available at: `http://www.combex.com/papers/darpa-report/index.html`.
33. Microsoft TechNet. User Account Control. Microsoft Corporation, 2007. Available at: `http://technet.microsoft.com/en-us/windowsvista/aa905113.aspx`.
34. Ollie Whitehouse. An example of why UAC prompts in Vista can't always be trusted. posted to Symantec Security Risks Weblog, February 20, 2007 05:00 AM, 2007. Available at: `http://www.symantec.com/enterprise/security_response/weblog/2007/02/an_example_of_why_uac_prompts.html`.
35. Ka-Ping Yee. Aligning security and usability. *IEEE Security and Privacy*, 2(5):48–55, September/October 2004.