Analysing the Information Flow Properties of Object-Capability Patterns

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. We consider the problem of detecting covert channels within security-enforcing object-capability patterns. Traditional formalisms for reasoning about the security properties of object-capability patterns require one to be aware, *a priori*, of all possible mechanisms for covert information flow that might be present within a pattern, in order to detect covert channels within it. We show how the CSP process algebra, and its model-checker FDR, can be applied to overcome this limitation.

1 Introduction

The *object-capability model* [9] is a security architecture for the construction of software systems that naturally adhere to the *principle of least authority* [9], a refinement of Saltzer and Schroeder's *principle of least privilege* [19]. Several current research projects, including secure programming languages like E [9], Joe-E [8] and Google's Caja [10], and microkernel operating systems like the Annex Capability Kernel [5] and seL4 [1], implement the object-capability model to provide platforms for cooperation in the presence of mutual suspicion.

Security properties are enforced in object-capability systems by deploying security-enforcing abstractions, called *patterns*, much in the same way that a program's ordinary functional properties are implemented by ordinary programming abstractions and design patterns. It is therefore very important to be able to understand precisely the security properties that an individual object-capability pattern does, and does not, enforce.

For systems in which confidentiality is a primary concern, we are most often interested in those security properties that capture the ways in which information may flow within them. In object-capability applications that involve confidentiality, the information flow properties of security-enforcing object-capability patterns are of vital importance. In particular, it is necessary to be able to detect the existence of covert channels within object-capability patterns.

Whilst the formal analysis of object-capability patterns has received some attention [20], the previous formalisms that were employed require all effects that are to be reasoned about to be explicitly included in any model of an object-capability pattern that is being analysed. Thus, in order for covert channels to be detected within an object-capability pattern, the mechanisms for covert

information propagation must be explicitly modelled. This requires one who wishes to detect covert channels in a pattern to be aware, *a priori*, of the possible mechanisms for covert information flow within it.

In this paper, we show how the CSP process algebra [14], and its model checker FDR [4], can be applied to model object-capability patterns and detect covert channels within them, without forcing the programmer to enumerate the mechanisms by which information may covertly propagate. We adopt CSP for modelling object-capability systems, as opposed to what others might consider to be a more natural formalism such as the π-calculus, because we can use FDR to automatically check our properties via CSP's formal theory of refinement which, as will become evident, is integral to our understanding of both object-capability systems and information flow within them.

We conclude this section by briefly explaining the object-capability model and the fragment of CSP used in this paper. Further details about CSP can be found in [14]. In Section 2, we explain how object-capability systems can be modelled in CSP. In doing so, we present an example model of a *Data-Diode* pattern, from [9], that is designed to allow data to flow from low-sensitivity objects to high-sensitivity ones, whilst preventing data propagating in the reverse direction. In Section 3, we give a general definition for information flow security for object-capability systems modelled in CSP and argue that the information flow property *Weakened RCFNDC for Compositions* [12], which can be automatically tested in FDR, is an appropriate test to apply to such systems. Applying this test to our model from Section 2, we find that it does indeed contain covert channels, before showing how to refine the model to an implementation that passes the test. The analysis here considers only a small instance of the Data-Diode pattern composed with a handful of other objects. Therefore, in Section 4, we show how to generalise our results to systems of arbitrary size in which objects may create arbitrary numbers of other objects, applying the theory of *data-independence* [6]. Finally, we conclude and consider related work in Section 5.

Some proofs are omitted but appear in [11]. Thanks to Bill Roscoe for useful discussions about data-independence, and to the anonymous reviewers.

The Object-Capability Model The *object-capability model* [9] is a model of computation and security that aims to capture the semantics of many actual object-based programming languages and capability-based systems, including all of those mentioned in Section 1. An *object-capability system* is an instance of the model and comprises just a collection of *objects*, connected to each other by *capabilities*. An object is a protected entity comprising state and code that together define its behaviour. An object's state includes both data and the capabilities it possesses. A capability, c, is an unforgeable object reference that allows its holder to send messages to the object it references by *invoking* c.

In an object-capability system, the only overt means for objects to interact is by sending messages to each other. Capabilities may be passed between objects only within messages. In practice, object o can pass one of its capabilities, c, directly to object p only by invoking a capability it possesses that refers to p,

including c in the invocation. This implies that capabilities can be passed only between objects that are connected, perhaps via intermediate objects.

Each object may expose a number of interfaces, known as *facets*. A capability that refers to an object, o, also identifies a particular facet of o. This allows the object to expose different functionality to different clients by handing each of them a capability that identifies a separate facet, for example.

An object may also create others. In doing so, it must supply any resources required by the newly created object, including its code and any data and capabilities it is to possess initially. Hence, a newly created object receives its first capabilities solely from its parent. When creating an object, the parent exclusively receives a capability to the child. Thus, an object's parent has complete control over those objects the child may come to interact with in its lifetime. This is the basis upon which mandatory security policies can be enforced [9].

In object-capability operating systems like seL4, each process may be thought of as a separate object. In object-capability languages like Caja, objects are akin to those from object-oriented languages; capabilities are simply object references.

CSP A system modelled in CSP comprises a set of concurrently executing *processes* that execute by performing *events*. Processes communicate by synchronising on common events, drawn from the set Σ of all visible events.

The process $STOP$ represents deadlock and cannot perform any events. The process $?a : A \rightarrow P_a$ is initially willing to perform all events from the set A and offers its environment the choice of which should be performed. Once a particular event, $a \in A$, has been performed, it behaves like the process P_a.

CSP allows multi-part events to be defined, where a dot is used to separate each part of an event. Suppose we define the set of events $\{\mathsf{plot}.x.y \mid x, y \in \mathbb{N}\}$. Then the process $\mathsf{plot}?x?y \rightarrow STOP$ offers all plot events whilst the process $\mathsf{plot}?x : \{1, \ldots, 5\}!3 \rightarrow STOP$ offers all events from $\{\mathsf{plot}.x.3 \mid x \in \{1, \ldots, 5\}\}$. $\{|c_1 . \ldots . c_k|\}$ denotes the set of events whose first k components are $c_1, \ldots, c_k$.

The process $P \,\Box\, Q$ can behave like either the process P or the process Q and offers its environment the initial events of both processes, giving the environment the choice as to which process it behaves like. The process $P \sqcap Q$ can also behave like either P or Q but doesn't allow the environment to choose which; instead, it makes this choice internally. $P \setminus A$ denotes the process obtained when P is run but all occurrences of the events in A are hidden from its environment.

The process $P \underset{A}{\parallel} Q$ runs the processes P and Q in parallel forcing them to synchronise on all events from the set A. The process $S = \Big\|_{1 \leq i \leq n} (P_i, A_i)$ is the *alphabetised parallel composition* of the n processes $P_1, \ldots, P_n$ on their corresponding *alphabets* $A_1, \ldots, A_n$. Each process P_i may perform events only from its alphabet A_i, and each event must be synchronised on by all processes in whose alphabet it appears. $P_1 \;{}_{A_1}\|_{A_2}\; P_2$ is equivalent to $\Big\|_{1 \leq i \leq 2} (P_i, A_i)$.

A process *diverges* when it performs an infinite number of internal τ events. A process *terminates* by performing the special termination event $\checkmark$. In this paper, we restrict our attention to processes that never diverge nor terminate.

Given a CSP process P, $traces(P)$ denotes the set that contains all finite sequences of visible events (including all prefixes thereof) that it can perform. A *stable-failure* is a pair (s, X) and denotes a process performing the finite sequence of events s and then reaching a *stable* state in which no internal τ events can occur, at which point all events from X are unavailable, *i.e.* X can be refused. We write $failures(P)$ for the set that contains all stable-failures of the process P. We write $divergences(P)$ for the set of traces of P after which it can diverge. For all of the processes P that we consider in this paper, $divergences(P) = \{\}$. For any divergence-free process P, $traces(P) = \{s \mid (s, X) \in failures(P)\}$.

CSP's standard denotational semantic model is the *failures-divergences model* [14]. Here a process P is represented by the two sets: $failures(P)$ and $divergences(P)$. One CSP process P is said to *failures-divergences refine* another Q, precisely when $failures(P) \subseteq failures(Q) \wedge divergences(P) \subseteq divergences(Q)$. In this case, we write $Q \sqsubseteq P$.

Sequences are written between angle brackets; $\langle\rangle$ denotes the empty sequence. $s \hat{} t$ denotes the concatenation of sequences s and t. $s \setminus H$ denotes the sequence obtained by removing all occurrence of events in the set H from the sequence s. $s \upharpoonright H$ denotes the sequence obtained by removing all non-H events from s.

2 Modelling Object-Capability Systems in CSP

In this section, we describe our approach to modelling object-capability systems in CSP. Note that we ignore the issue of object creation for now. This will be handled later on in Section 4.

We model an object-capability system *System* that comprises a set *Object* of objects as the alphabetised parallel composition of a set of processes $\{behaviour(o) \mid o \in Object\}$ on their corresponding alphabets $\{\alpha(o) \mid o \in Object\}$. So $System = \|_{o \in Object} (behaviour(o), \alpha(o))$.

The facets of each object $o \in Object$ are denoted $facets(o)$. We restrict our attention to those well-formed systems in which $facets(o) \cap facets(p) \neq \{\} \Rightarrow o = p$. Recall that an individual capability refers to a particular facet of a particular object. Hence, we define the set $Cap = \bigcup\{facets(o) \mid o \in Object\}$ that contains all entities to which capabilities may refer.

The events that each process $behaviour(o)$ can perform represent it sending and receiving messages to and from other objects in the system. We define events of the form $f_1.f_2.op.arg$ to denote the sending of a message from the object with facet f_1 to facet f_2 of the object with this facet, requesting it to perform operation op, passing the argument arg and a *reply* capability f_1, which can be used later to send back a response. Here $f_1, f_2 \in Cap$. Arguments are either capabilities, data or the special value null, so $arg \in Cap \cup Data \cup \{\mathsf{null}\}$, for some set $Data$ of data. An operation op comes from the set $\{\mathsf{Call}, \mathsf{Return}\}$. These operations model a call/response remote procedure call sequence in an object-capability operating system or a method call/return in an object-capability language.

The alphabet of each object $o \in Object$ contains just those events involving o. Hence, $\alpha(o) = \{|f_1.f_2 \mid f_1, f_2 \in Cap \wedge (f_1 \in facets(o) \vee f_2 \in facets(o))|\}$.

We require that the process $behaviour(o)$ representing the behaviour of each object $o \in Object$ adheres to the basic rules of the object-capability model, such as not being able to use a capability it has not legitimately acquired. We codify this by defining the most general and nondeterministic process that includes all permitted behaviours (and no more) that can be exhibited by an object o. Letting $facets = facets(o)$ denote the set that comprises o's facets, and $caps \subseteq Cap$ and $data \subseteq Data$ denote the sets of capabilities and data that o initially possesses, the most general process that includes all behaviours permitted by the object-capability model that o may perform is denoted $Untrusted(facets, caps, data)$.

$$
\begin{aligned}
&Untrusted(facets, caps, data) = \\
&\quad \left(\begin{array}{l} ?me : facets?c : caps \cup facets?op?arg : caps \cup data \cup \{\mathsf{null}\} \rightarrow \\ \qquad Untrusted(facets, caps, data) \ \Box \\ ?from : Cap - facets?me : facets?op?arg \rightarrow \\ \qquad Untrusted(facets, caps \cup (\{arg, from\} \cap Cap), data \cup (\{arg\} \cap Data)) \end{array} \right) \\
&\quad \sqcap STOP.
\end{aligned}
$$

This object can invoke only those capabilities $c \in caps \cup facets$ that it possesses. In doing so it requests an operation op, and may include only those arguments $arg \in caps \cup data \cup \{\mathsf{null}\}$ it has, along with a reply capability $me \in facets$ to one of its facets. Having done so, it returns to its previous state.

This object can also receive any invocation from any other, where the reply capability included in the invocation is $from \in Cap - facets$, to one of its facets $me \in facets$, requesting an arbitrary operation op, and containing an arbitrary argument arg. If such an invocation occurs, the object may acquire the reply capability $from$, as well as any capability or datum arg in the argument. This process may also deadlock at any time, making it maximally nondeterministic.

The behaviour $behaviour(o)$ of an object $o \in Object$, whose initial capabilities and data are $caps(o)$ and $data(o)$ respectively, is then valid if and only if all behaviours it contains are present in $Untrusted(facets(o), caps(o), data(o))$. This leads to the following definition of a valid object-capability system.

Definition 1 (Object-Capability System). *An* object-capability system *is a tuple* $(Object, behaviour, facets, Data)$*, where* $Object$*,* $behaviour$*,* $facets$ *and* $Data$ *are as discussed above and, letting* $Cap = \bigcup\{facets(o) \mid o \in Object\}$*, there exist functions* $caps : Object \rightarrow \mathbf{P}\, Cap$ *and* $data : Object \rightarrow \mathbf{P}\, Data$ *that assign minimal initial capabilities and data to each object so that, for each* $o \in Object$*,* $Untrusted(facets(o), caps(o), data(o)) \sqsubseteq behaviour(o)$*.*

2.1 An Example Pattern

We illustrate these concepts by modelling the Data-Diode pattern [9, Figure 11.2], which is designed to allow low-sensitivity objects to send data to to high-sensitivity ones whilst preventing information flowing the other way.

A *data-diode* is an object that has two facets, a read-facet and a write-facet[1]. It stores a single datum and begins life holding some initial value. In-

[1] It is unclear whether the read and write interfaces should be implemented as facets of a single object or as forwarding objects of a composite object. We choose the

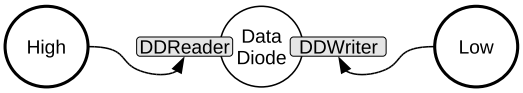

Fig. 1. A system in which to analyse the Data-Diode pattern. Bold circles indicate objects with arbitrary behaviour.

voking its read-facet causes it to return its current contents. Invoking its write-facet with an argument causes it to replace its current contents with the argument. We model a data-diode with read-facet *readme* and write-facet *writeme* that initially contains the datum *val* from the set *Data* as the CSP process $ADataDiode(readme, writeme, val)$, defined as follows.

$$
\begin{aligned}
&ADataDiode(readme, writeme, val) = \\
&\quad ?from : Cap - \{readme, writeme\}!readme!\mathsf{Call}!\mathsf{null} \rightarrow \\
&\qquad readme!from!\mathsf{Return}!val \rightarrow ADataDiode(readme, writeme, val) \ \Box \\
&\quad ?from : Cap - \{readme, writeme\}!writeme!\mathsf{Call}?newVal : Data \rightarrow \\
&\qquad writeme!from!\mathsf{Return}!\mathsf{null} \rightarrow ADataDiode(readme, writeme, newVal).
\end{aligned}
$$

Observe that this process passes *Data* items only, refusing all *Cap* arguments.

To analyse this pattern, we instantiate it in the context of the object-capability system *System* depicted in Figure 1. Here, we see a data-diode, DataDiode, with read- and write-facets DDReader and DDWriter respectively. An arbitrary high-sensitivity object High has a capability to the data-diode's read-facet, allowing it to read data written by an arbitrary low-sensitivity object Low, which has a capability to the data-diode's write-facet. High and Low possess the data HighDatum and LowDatum respectively.

Let $Object = \{\mathsf{High}, \mathsf{DataDiode}, \mathsf{Low}\}$, $HighData = \{\mathsf{HighDatum}\}$, $LowData = \{\mathsf{LowDatum}\}$, $Data = HighData \cup LowData$, $facets(\mathsf{DataDiode}) = \{\mathsf{DDReader}, \mathsf{DDWriter}\}$ and $facets(other) = \{other\}$ for $other \neq \mathsf{DataDiode}$. The process *System* is then defined as explained earlier using the behaviours:

$$
\begin{aligned}
&behaviour(\mathsf{DataDiode}) = ADataDiode(\mathsf{DDReader}, \mathsf{DDWriter}, \mathsf{null}), \\
&behaviour(\mathsf{High}) = \\
&\quad Untrusted(facets(\mathsf{High}), facets(\mathsf{High}) \cup \{\mathsf{DDReader}\}, HighData), \\
&behaviour(\mathsf{Low}) = Untrusted(facets(\mathsf{Low}), facets(\mathsf{Low}) \cup \{\mathsf{DDWriter}\}, LowData).
\end{aligned}
$$

3 Information Flow in Object-Capability Patterns

Performing some basic refinement checks in FDR, which test whether certain events cannot occur in *System*, reveals that Low cannot obtain any *HighData* but that High can obtain LowDatum in this system. We now consider how to test

former option at this point and will explore the latter in Section 3.1.

whether, despite preventing this overt flow of data, DataDiode might provide a covert channel from High to Low. We will argue that the correct property to apply to *System* is *Weakened RCFNDC for Compositions*, introduced in [12].

Information flow properties have been well-studied in the context of process algebras, including CSP (*e.g.* [3, 2, 17, 7]). The obvious approach would take one of these properties and apply it to the process *behaviour*(DataDiode) to see whether it allows information to flow from its high interface DDReader to its low interface DDWriter.

However, this approach doesn't take into account the constraints imposed by the object-capability model on the objects like High and Low that may interact with DDReader and DDWriter. This is because these constraints are not reflected in *behaviour*(DataDiode) but are instead imposed upon *behaviour*(High) and *behaviour*(Low). For example, observe that initially the process *behaviour*(DataDiode) can perform the event High.DDWriter.Call.null; however, this event cannot be performed in *System* because it can occur there only when both High and DataDiode are willing to perform it, and *behaviour*(High) cannot perform it initially because High does not initially possess a capability to DDWriter. In order to get accurate results, therefore, one needs to analyse the entire system *System*, using an appropriate information flow property.

Recall that the processes *behaviour*(High) and *behaviour*(Low), which both instantiate the process *Untrusted*, are purposefully highly nondeterministic, in order to ensure that each is as general as possible. This makes the entire system *System* very nondeterministic. It has long been recognised that many standard information flow properties suffer from the so-called "refinement paradox" in which a property holds for a system but can be violated by one of the system's refinements. The refinements of a system capture the ways in which nondeterminism can be resolved in it. The refinement paradox is dangerous because it allows a nondeterministic system to be deemed secure when, under some resolution of the system's nondeterminism, it may actually be insecure [7].

A fail-safe way to avoid the refinement paradox is to apply an information flow property that is *refinement-closed* [7]. A property is refinement-closed when, for every process P, it holds for P only if it holds for all P's refinements.

While we want to avoid the refinement paradox, refinement-closed properties are too strong for our purposes. This is because the refinements of a parallel composition include those in which the resolution of nondeterminism in one component can depend on activity that occurs within the system that the component cannot overtly observe.

For example, *System* is refined by a process that has the trace $\langle \mathsf{High.High.Call.High}, \mathsf{Low.Low.Call.null} \rangle$ but also has the stable-failure $(\langle\rangle, \{\mathsf{Low.Low.Call.null}\})$. This refinement means that *System* fails a number of refinement-closed information flow properties, *e.g.* Roscoe's *Lazy Independence* [14, Section 12.4] and Lowe's *Refinement-Closed Failures Non-Deducibility on Compositions* [7]. These two behaviours arise because of the nondeterminism in Low: initially Low may either perform Low.Low.Call.null or may refuse it, depending on how this nondeterminism is resolved. In the trace

above, where High performs the event High.High.Call.High, Low's nondeterminism is resolved such that Low.Low.Call.null occurs; while in the stable-failure, where High doesn't perform its event, this nondeterminism in Low is resolved the other way. The resolution of the nondeterminism in Low here thus depends on whether High has performed its event, in which it interacts with just itself.

A system that exhibits both of these behaviours therefore allows High's interactions with just itself to somehow influence Low. In such systems it is impossible to talk sensibly about the information flow properties of the Data-Diode pattern.

We see that in general, one cannot talk sensibly about the information flow properties of object-capability patterns without assuming that the only way for one object to directly influence another is by sending it a message or receiving one from it, since it is only overt message passing that any pattern can hope to control. Thus, in any system, we assume that the resolution of nondeterminism in any object can be influenced only by the message exchanges in which it has partaken before the nondeterminism is resolved.

Without specifying *how* the nondeterminism in any object may be resolved after it has engaged in some sequence s of message exchanges, this therefore implies that whenever it performs s, the nondeterminism should be resolved *consistently* [12]. Two resolutions of the nondeterminism in a process after it has performed s are inconsistent when it can perform some event e in one but refuse e in the other. Under this definition, the two different resolutions of the nondeterminism in Low above, depending on whether High has performed its event that doesn't involve Low, are inconsistent: in each case, Low performs/refuses the event Low.Low.Call.null after performing no others.

We therefore confine ourselves to the ways of resolving the nondeterminism in each object in which this kind of inconsistency does not arise. Note that these are precisely the *deterministic* refinements of each object, under the standard definition of determinism for CSP processes.

Definition 2 (Determinism). *A divergence-free process P is said to be* deterministic*, written $det(P)$, iff $\nexists\, s, e \bullet s \mathbin{\hat{}} \langle e \rangle \in traces(P) \wedge (s, \{e\}) \in failures(P)$.*

With this in mind, we seek an information flow property that holds for a system $System = \Big\|_{o \in Object} (behaviour(o), \alpha(o))$ just when those refinements of *System*, in which the nondeterminism in each object is resolved to produce a deterministic process, are deemed secure. Any such *deterministic componentwise refinement* may be written as $System' = \Big\|_{o \in Object} (b_o, \alpha(o))$ where $\forall\, o \in Object \bullet behaviour(o) \sqsubseteq b_o \wedge det(b_o)$. Let $DCRef(System)$ denote the set of all deterministic componentwise refinements of *System*. Any such refinement $System' \in DCRef(System)$ will itself be deterministic [14]. Many information flow properties, which might otherwise disagree, agree when applied to deterministic processes. Hence, given any such property *Prop*, we arrive at the following definition of information flow security for object-capability systems.

Definition 3. *An object-capability system captured by the CSP process $System = \Big\|_{o \in Object} (behaviour(o), \alpha(o))$ is secure under componen-*

twise refinement with respect to the information flow property Prop iff $\forall\, System' \in DCRef(System) \bullet Prop(System')$.

3.1 Testing Information Flow

The information flow property *Weakened RCFNDC for Compositions* [12] is equivalent to Definition 3 when *Prop* is Lowe's *Refinement-Closed Failures Non-Deducibility on Compositions* [7] (RCFNDC). RCFNDC is equivalent when applied to deterministic processes to a number of standard information flow properties, including [16] at least all those that are no stronger than Roscoe's *Lazy Independence* [14, Section 12.4] and no weaker than Ryan's traces formulation of noninterference [18, Equation 1]. Therefore, we adopt Weakened RCFNDC and its associated automatic refinement check [12] to test for information flow here.

Like similar information flow properties, given two sets H and L that partition the alphabet of a system, Weakened RCFNDC tests whether the occurrence of events from H can influence the occurrence of events from L. In [12], it is shown that any divergence-free alphabetised parallel composition $S = \Big\|_{1 \le i \le n} (P_i, A_i)$ satisfies Weakened RCFNDC, written $WRCFNDC(S)$, iff:

$$\begin{array}{l} \nexists\, s, l \bullet s \upharpoonright H \neq \langle\rangle \wedge l \in L \wedge \\ \quad \left(\begin{array}{l} s \mathbin{\hat{}} \langle l \rangle \in traces(S) \wedge s \setminus H \in traces(S) \wedge \\ \exists\, i \bullet l \in A_i \wedge s \upharpoonright A_i \neq s \setminus H \upharpoonright A_i \wedge (s \setminus H \upharpoonright A_i, \{l\}) \in failures(P_i) \end{array} \right) \vee \\ \quad \left(\begin{array}{l} s \setminus H \mathbin{\hat{}} \langle l \rangle \in traces(S) \wedge s \in traces(S) \wedge \\ \exists\, i \bullet l \in A_i \wedge s \upharpoonright A_i \neq s \setminus H \upharpoonright A_i \wedge (s \upharpoonright A_i, \{l\}) \in failures(P_i) \end{array} \right). \end{array} \quad (1)$$

Let $H = \{|h.\mathsf{DDReader}, \mathsf{DDReader}.h, h.h' \mid h, h' \in facets(\mathsf{High})|\}$ denote the set of events that represent High interacting with DDReader and itself. Similarly let $L = \{|l.\mathsf{DDWriter.Call}.arg, \mathsf{DDWriter}.l.\mathsf{Return.null}, l.l' \mid l, l' \in facets(\mathsf{Low})|\}$. Then a refinement check in FDR reveals that *System* from Section 2.1 can perform no events outside of $H \cup L$. This implies, for example, that neither High nor Low can obtain a capability to the other. Therefore, H and L partition the effective alphabet of *System*.

Applying the refinement check for Weakened RCFNDC to *System* with these definitions of H and L, using FDR, reveals that Weakened RCFNDC doesn't hold. Interpreting the counter-example returned from FDR, we see that *System* can perform the trace $\langle \mathsf{Low.DDWriter.Call.LowDatum} \rangle$ but also has the failure $(\langle \mathsf{High.DDReader.Call.null} \rangle, \{\mathsf{Low.DDWriter.Call.LowDatum}\})$. This indicates that initially Low can invoke DDWriter but that if High invokes DDReader, it can cause Low's invocation to be refused. This occurs because DataDiode cannot service requests from High and Low at the same time. This constitutes a clear covert channel, since High can signal to Low by invoking DDReader which alters whether Low's invocation is accepted.

Low may be unable to observe this covert channel in some object-capability systems, *e.g.* those in which a sender of a message is undetectably blocked until the receiver is ready to receive it. For this kind of system, one might wish to replace *Prop* with another property, such as Focardi and Gorrieri's *Traces NDC* [3],

that detects only when high events can cause low events to occur, rather than also detecting when they can prevent them from occurring as happens in the counter-example above. Modifying Weakened RCFNDC to do so simply involves removing the second disjunct from Equation 1. However, we choose to make the conservative assumption that this counter-example represents a valid fault.

Correcting the fault here involves modifying the data-diode implementation so that its interfaces for writing and reading, DDWriter and DDReader, can be used simultaneously. We do so by promoting these interfaces from being facets of a single process to existing as individual processes in their own right. These processes simply act now as proxies that forward invocations to the facets of an underlying *ADataDiode* process, as depicted in Figure 2.

The behaviour of a proxy *me* that forwards invocations it receives using the capability *target* is given by the process $AProxy(me, target)$ defined as follows.

$$\begin{aligned} AProxy(me, target) = {} & ?from : Cap - \{me\}!me!\mathsf{Call}?arg : Data \cup \{\mathsf{null}\} \rightarrow \\ & me!target!\mathsf{Call}!arg \rightarrow target!me!\mathsf{Return}?res : Data \cup \{\mathsf{null}\} \rightarrow \\ & me!from!\mathsf{Return}!res \rightarrow AProxy(me, target). \end{aligned}$$

The data-diode is now a *composite* of three entities, DDReader, DDWriter and DataDiode, and as such is referred to as DDComposite. We model the system depicted in Figure 2 as an object-capability system comprising the objects from $Object = \{\mathsf{High}, \mathsf{DDComposite}, \mathsf{Low}\}$, where $facets(\mathsf{DDComposite}) = \{\mathsf{DDReader}, \mathsf{DDWriter}, \mathsf{DDR}, \mathsf{DDW}\}$ and, letting $R = \{|\mathsf{DDReader}.x, x.\mathsf{DDReader} \mid x \in facets(\mathsf{DDComposite}) - \{\mathsf{DDReader}\}|\}$, $W = \{|\mathsf{DDWriter}.x, x.\mathsf{DDWriter} \mid x \in facets(\mathsf{DDComposite}) - \{\mathsf{DDWriter}\}|\}$, $DD = ADataDiode(\mathsf{DDR}, \mathsf{DDW}, \mathsf{null})$ and the other definitions be as before,

$$behaviour(\mathsf{DDComposite}) = \left((AProxy(\mathsf{DDReader}, \mathsf{DDR}) \underset{R}{\parallel} DD) \underset{W}{\parallel} AProxy(\mathsf{DDWriter}, \mathsf{DDW}) \right) \setminus (R \cup W).$$

DDComposite is formed by taking the two proxies, DDReader and DDWriter, and composing them in parallel with DataDiode, whose read- and write-interfaces are now DDR and DDW respectively. Notice that we then hide the internal communications within DDComposite since these are not visible to its outside environment and it is unclear how to divide these events between the sets H and L. FDR can be used to check that this system, *System*, satisfies Definition 1.

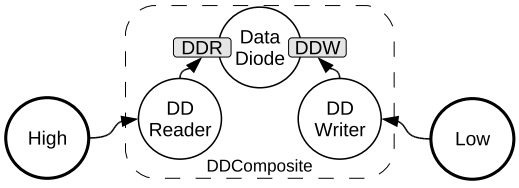

Fig. 2. An improved Data-Diode implementation.

Performing the appropriate refinement checks in FDR reveal that High can acquire LowDatum but Low cannot acquire any *HighData*, and that *System* can perform no events outside of $H \cup L$, as before. FDR reveals that Weakened RCFNDC holds for *System*. Hence, we are unable to detect any covert channels in this model of the improved Data-Diode implementation.

4 Generalising the Results

We have verified this improved Data-Diode implementation in the context of only a handful of other objects and in the absence of object creation. In this section, we show how to generalise our analysis to all systems that have the form of Figure 3, and have arbitrary *HighData* and *LowData*. Here, the objects within each cloud can be interconnected in any way whatsoever; however, the only capability to an object outside of the high object cloud that each high object may possess is DDReader. The same is true for the low objects and DDWriter. This figure captures all systems containing an arbitrary number of high- and low-sensitivity objects and, thus, all those in which each object may create arbitrary numbers of others that share its level of sensitivity.

Roughly, the approach we take is to show that the improved system analysed in the previous section is a *safe abstraction* of all systems captured by Figure 3, such that if the safe abstraction is deemed secure then so will all of the systems it abstracts. For one system $System' = \Big\|_{o \in Object'} (behaviour'(o), \alpha'(o))$ to be a safe abstraction of another $System = \Big\|_{o \in Object} (behaviour(o), \alpha(o))$, we require that if $System'$ is deemed secure, then so must *System*.

Recall that, by Definition 3, *System* is secure iff $Prop(System_D)$ holds for all $System_D \in DCRef(System)$ for some information flow property *Prop*. We therefore insist that in order for $System'$ to be a safe abstraction of *System*, that each $System_D$ is also present in $DCRef(System')$.

Definition 4 (Safe Abstraction). *Given any* $System'$ *and System as above,* $System'$ *is a* safe abstraction *of System iff* $DCRef(System) \subseteq DCRef(System')$.

We now show that each system *System* captured by Figure 3 can be safely abstracted by a system $System'$ of the form of Figure 2. We form $System'$ by taking each cloud of objects in *System* and *aggregating* all of the objects in the

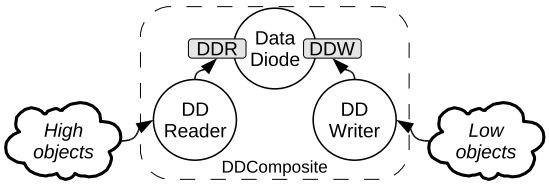

Fig. 3. Generalising the results.

cloud into a single object in $System'$. In order to be a proper aggregation, each object from $System'$ must have all facets, capabilities, data and behaviours of all the objects from $System$ that it aggregates. We formally capture that $System'$ is an aggregation of $System$ via a surjection $Abs : Object \to Object'$ that maps each object of $System$ to the object that aggregates it in $System'$.

Definition 5 (Aggregation). *Let* $(Object, behaviour, facets, Data)$ *and* $(Object', behaviour', facets', Data)$ *be two object-capability systems with identical sets of data, captured by* $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ *and* $System' = \|_{o \in Object'}(behaviour'(o), \alpha'(o))$ *respectively. Then the second is an* aggregation *of the first when there exists some surjection* $Abs : Object \to Object'$ *such that for all* $o' \in Object'$, $facets'(o') = \bigcup\{facets(o) \mid o \in Abs^{-1}(o')\}$ *and*

$$\begin{array}{l} \forall\, s \in traces(System) \bullet \forall\, X \in \mathbf{P}\, \Sigma \bullet \\ \quad (s \upharpoonright \alpha'(o'), X) \in failures(\|_{o \in Abs^{-1}(o')}(behaviour(o), \alpha(o))) \Rightarrow \\ \quad\quad (s \upharpoonright \alpha'(o'), X) \in failures(behaviour'(o')), \end{array}$$

where $Abs^{-1}(o') = \{o \mid o \in Object \wedge Abs(o) = o'\}$.

The proof of the following theorem requires some technical results beyond the scope of this paper; given limitations on space, it can be found in [11].

Theorem 1. *Let System and* $System'$ *capture two object-capability systems as stated in Definition 5. Then if* $System'$ *is an aggregation of System, it is also a safe abstraction of System.*

We claim that any finite collection $K \subseteq Object$ of objects $\|_{o \in K}(behaviour(o), \alpha(o))$, can be aggregated by a single object with behaviour $Untrusted(\bigcup_{o \in K} facets(o), \bigcup_{o \in K} caps(o), \bigcup_{o \in K} data(o))$ that has all of their capabilities, data and facets. Briefly, by Definition 1, $behaviour(o) \sqsubseteq Untrusted(facets(o), caps(o), data(o))$ for each $o \in K$ for some sets $caps(o)$ and $data(o)$ of capabilities and data that it possesses initially. Further $Untrusted(facets(o) \cup facets(o'), caps(o) \cup caps(o'), data(o) \cup data(o')) \sqsubseteq Untrusted(facets(o), caps(o), data(o)) \;{}_{\alpha(o)}\|_{\alpha(o')}\; Untrusted(facets(o'), caps(o'), data(o'))$. The claim then follows by induction on the size of K.

So consider any system that has the form of Figure 3 and let T denote the facets of the high objects, U the facets of the low objects and $V = HighData \cup LowData$, *i.e.* $V = Data$. Then this system can be safety abstracted by a system $System_{T,U,V}$ of the form of Figure 2 in which $facets(High) = T$, $facets(\mathsf{Low}) = U$, $HighData = V$ and $LowData = V$, so that $Data = V$. Notice that we allow High and Low to both possess all data in the safe abstraction in order to obtain maximum generality. If we can show that $System_{T,U,V}$ is secure for all non-empty choices of T, U and V, by Theorem 1, we can conclude that the improved Data-Diode implementation is secure in all systems captured by Figure 3 with arbitrary $HighData$ and $LowData$.

The theory of *data-independence* [6] can be applied to show that a property *Prop* holds of a process P_T, parameterised by some set T, for all non-empty choices of T, if $Prop(P_T)$ can be shown for all non-empty T of size N or less, for some N. N is called the data-independence *threshold* for T for $Prop(P_T)$. The theory requires that P_T be *data-independent* in T, meaning roughly that P_T handles members of the type T uniformly, not distinguishing one particular value of T from another. We apply data-independence theory to show that thresholds of size 1, 2 and 2 for T, U and V respectively are sufficient to demonstrate the security of $System_{T,U,V}$ for all non-empty choices of each set.

We will use the following standard result. Let P_T be a process that is data-independent in some set T and satisfies **NoEqT** for T, meaning that it never needs to test two values of T for equality. Let ϕ be a surjection whose domain is T, where we write $\phi(T)$ for $\{\phi(t)|t \in T\}$ and $\phi^{-1}(X)$ for $\{y \mid y \in T \wedge \phi(y) \in X\}$. Then [6, Theorem 4.2.2], lifting ϕ to events and traces,

$$\{(\phi(s), X) \mid (s, \phi^{-1}(X)) \in failures(P_T)\} \subseteq failures(P_{\phi(T)}). \tag{2}$$

Theorem 2. *Let* $S_T = \Big\|_{1 \leq i \leq n} (P_{T,i}, A_{T,i})$ *be an alphabetised parallel composition, whose components and alphabets are polymorphically parameterised by some set* T*, such that* S_T *and each* $P_{T,i}$ *are data-independent in* T *and satisfy* **NoEqT** *for* T*. Also let* H_T *and* L_T *be two sets polymorphically parameterised by* T *that partition the alphabet of* S_T *for all non-empty* T*. Let* W *denote the maximum number of distinct elements of* T *that appear in any single event from* L_T*. Then* $W + 1$ *is a sufficient data-independence threshold for* T *for* $WRCFNDC(S_T)$*.*

Proof. Assume the conditions of the theorem. Suppose for some T with size greater than W, S_T fails Weakened RCFNDC for H_T and L_T. Then let $\tilde{T} = \{\tilde{t}_0, \ldots, \tilde{t}_W\}$ for fresh elements $\tilde{t}_0, \ldots, \tilde{t}_W$. We show that $S_{\tilde{T}}$ fails Weakened RCFNDC for $H_{\tilde{T}}$ and $L_{\tilde{T}}$.

Let $\phi : T \to \tilde{T}$ be a surjection; we fix the choice of ϕ below. Lift ϕ to events by applying ϕ to all components of type T. Then ϕ maps an event in the alphabet of S_T to an event in the alphabet of $S_{\tilde{T}}$. Also, lifting ϕ to sets of events, $\phi(A_{T,i}) = A_{\tilde{T},i}$ for $1 \leq i \leq n$, $\phi(H_T) = H_{\tilde{T}}$ and $\phi(L_T) = L_{\tilde{T}}$.

Observe that $S_{\phi(T)} = S_{\tilde{T}}$. So, by Equation 2, the presence of certain behaviours in S_T implies the presence of related behaviours in $S_{\tilde{T}}$. Recall the characterisation of Weakened RCFNDC from Equation 1. Suppose S_T fails the first disjunct of Equation 1 for H_T and L_T. We show that $S_{\tilde{T}}$ fails this disjunct for $H_{\tilde{T}}$ and $L_{\tilde{T}}$. The second disjunct is handled similarly. Then there exists some s, l and $i \in \{1, \ldots, n\}$ such that

$$s \upharpoonright H_T \neq \langle\rangle \wedge l \in L_T \wedge s \mathbin{\hat{}} \langle l \rangle \in traces(S_T) \wedge s \setminus H_T \in traces(S_T) \wedge$$
$$l \in A_{T,i} \wedge s \upharpoonright A_{T,i} \neq s \setminus H_T \upharpoonright A_{T,i} \wedge (s \setminus H_T \upharpoonright A_{T,i}, \{l\}) \in failures(P_{T,i}).$$

Let $t_0, \ldots, t_{k-1}$ be the distinct members of T that appear in l. Then $k \leq W$. Choose $\phi(t_i) = \tilde{t}_i$ for $0 \leq i \leq k-1$ and let $\phi(t) = \tilde{t}_k$ for all other $t \in T - \{t_0, \ldots, t_{k-1}\}$. Let $\tilde{s} = \phi(s)$ and $\tilde{l} = \phi(l)$. Then $\tilde{s} \upharpoonright \tilde{H} \neq \langle\rangle \wedge \tilde{l} \in \tilde{L} \wedge \tilde{l} \in A_{\tilde{T},i} \wedge \tilde{s} \upharpoonright A_{\tilde{T},i} \neq \tilde{s} \setminus \tilde{H} \upharpoonright A_{\tilde{T},i}$. Applying Equation 2 to S_T, we have

$\tilde{s}^\frown\langle\tilde{l}\rangle \in traces(S_{\tilde{T}}) \wedge \tilde{s}\backslash\tilde{H} \in traces(S_{\tilde{T}})$. Further, $\{l\} = \phi^{-1}(\{\tilde{l}\})$ by construction. So, applying Equation 2 to $P_{T,i}$, we obtain $(\tilde{s}\setminus\tilde{H}\upharpoonright A_{\tilde{T},i}, \{\tilde{l}\}) \in failures(P_{\tilde{T},i})$. □

Set $H_{T,U,V} = \{|t.\mathsf{DDReader}, \mathsf{DDReader}.t, t.t' | t, t' \in T|\}$ and $L_{T,U,V} = \{|u.\mathsf{DDWriter}.\mathsf{Call}.d, \mathsf{DDWriter}.u.\mathsf{Return}.\mathsf{null}, u.u' \mid u, u' \in U, d \in Data \cup \{\mathsf{null}\}|\}$. Then $System_{T,U,V}$ and all of its components are data-independent in T, U and V and satisfy **NoEqT** for each.

Applying Equation 2, it is easily shown that $H_{T,U,V}$ and $L_{T,U,V}$ partition the alphabet of $System_{T,U,V}$ for all non-empty choices of T, U and V, if they do so when each of these sets has size 1. FDR confirms the latter to be true.

To verify Weakened RCFNDC, Theorem 2 suggests thresholds for T, U and V of 1, 4 and 2 respectively. This threshold for U arises from events in $L_{T,U,V}$ of the form $u.u'.op.u''$ for $u, u', u'' \in U$.

In fact, in the proof of Theorem 2, l is necessarily an event in the alphabet of a process that can perform both H_T and L_T events. Hence, we can strengthen this theorem to take W to be the maximum number of distinct values of type T in all such events in L_T. For $System_{T,U,V}$, this means that all events from $\{|u.u' \mid u, u' \in U|\}$ can be excluded when calculating the threshold for U, reducing it to 2.

The most expensive of the 4 tests implied by these thresholds examines about 6 million state-pairs, taking around 4 minutes to compile and complete on a desktop PC; the others are far cheaper. All tests pass, generalising our results.

5 Conclusion and Related Work

We have shown how to apply CSP and FDR to automatically detect covert channels in security-enforcing object-capability patterns without forcing the programmer to specify the mechanisms by which information may propagate covertly.

Our approach couples the objects that implement a pattern with arbitrary, *Untrusted*, high- and low-sensitivity objects that exhibit all behaviours permitted by the object-capability model. This has the added advantage that we can compare how a pattern functions in different kinds of object-capability system, such as single-threaded versus concurrent systems, by simply refining the definition of the *Untrusted* process. Investigating how the information flow properties of patterns are affected by changing the context in which they are deployed is an obvious avenue for future work.

The assumption that objects affect each other only by passing messages means that our analysis cannot be applied to *timed* systems in which objects have access to a global clock, for instance. Extending this work to cover such systems may allow us to detect possible timing channels that may exist in them.

Spiessens' [20] is the only prior work of which we are aware that examines the security properties of object-capability patterns. The ideas of *safe abstraction* and *aggregation* defined in Section 4 were heavily inspired by similar ideas in [20]. Spiessens' formalism has the advantage of not requiring the use of data-independence arguments to generalise analyses of small systems to large systems. On the other hand, our approach, unlike Spiessens', can detect covert channels in

a pattern without forcing the programmer to specify the means by which information can propagate covertly. Instead, these means are captured by information flow properties that can be applied to any pattern being analysed.

The notion of aggregation is also similar to (the inverse of) van der Meyden's *architectural refinement* [21]. Finally, data-independence theory has been applied before to generalise analyses of small systems to larger systems, including to the analysis of cryptographic protocols [15] and intrusion detection systems [13].

References

1. D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In *Proceedings of VSTTE '08*, pages 99–114. Springer, 2008.
2. R. Focardi. Comparing two information flow security properties. In *Proceedings of CSFW '96*, pages 116–122. IEEE Computer Society, 1996.
3. R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.
4. Formal Systems (Europe), Limited. *FDR2 User Manual*, 2005.
5. D. Grove, T. Murray, C. Owen, C. North, J. Jones, M. R. Beaumont, and B. D. Hopkins. An overview of the Annex system. In *Proceedings of ACSAC '07*, 2007.
6. R. S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. D.Phil. thesis, Oxford University Computing Laboratory, 1999.
7. G. Lowe. On information flow and refinement-closure. In *Proceedings of the Workshop on Issues in the Theory of Security (WITS '07)*, 2007.
8. A. M. Mettler and D. Wagner. The Joe-E language specification, version 1.0. Technical Report EECS-2008-91, University of California, Berkeley, August 2008.
9. M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.
10. M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript (draft), 2008.
11. T. Murray. *Analysing the Security Properties of Object-Capability Patterns*. D.Phil. thesis, University of Oxford, 2010. Forthcoming.
12. T. Murray and G. Lowe. On refinement-closed security properties and nondeterministic compositions. In *Proceedings of AVoCS '08*, pages 49–68, 2009.
13. G. T. Rohrmair and G. Lowe. Using data-independence in the analysis of intrusion detection systems. *Theoretical Computer Science*, 340(1):82–101, 2005.
14. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
15. A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *J. Comput. Secur.*, 7(2-3):147–190, 1999.
16. A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proceedings of CSFW '99*, page 228. IEEE Computer Society, 1999.
17. P. Ryan and S. Schneider. Process algebra and non-interference. *Journal of Computer Security*, 9(1/2):75–103, 2001.
18. P. Y. A. Ryan. A CSP formulation of non-interference and unwinding. *IEEE Cipher*, pages 19–30, Winter 1991.
19. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1208–1308, September 1975.
20. A. Spiessens. *Patterns of Safe Collaboration*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, February 2007.
21. R. van der Meyden. Architectural refinement and notions of intransitive noninterference. In *Proceedings of ESSoS 2009*, pages 60–74. Springer, 2009.