Secure Information Flow in Orc (DRAFT)

Adrian Quark

May 19, 2009

Abstract

Secure information flow attempts to verify that programs do not leak information to unauthorized third parties. Previous approaches to secure information flow have considered classical sequential languages and concurrent languages based on channels. In this work, we demonstrate how techniques from secure information flow can be used to verify security properties of *structured concurrent workflows* expressed in the Orc calculus. Unlike channel-based concurrent languages, Orc imposes structure on the flow of information between processes, enabling more straightforward analysis of some programs. Orc also relies heavily on non-termination to manage control flow, requiring a more sophisticated analysis of termination properties.

1 Introduction

It is a given that some computer programs need to process sensitive and confidential data. How can we ensure that such programs are trustworthy? The most straightforward approach, to totally isolate these programs from their environment, is not practical; today's applications are made of diverse, often distributed, components which process different levels of sensitive information. Totally isolating any one of these components from the others is impossible. Therefore it is important to be able to verify that each component shares information securely, only with parties who are authorized to access it. The approach discussed in this paper, known as *secure information flow* and popularized by Denning and Denning [3], attempts to prove that the flow of information in a program is secure by proving that high-security information never becomes visible on low-security outputs.

Information flow security is applicable not just to computer programs, but to any process which deals with sensitive information and can be modeled formally. This paper focuses specifically on the application of information flow security to workflows. Workflows are business processes which describe the flow of information and delegation of work within an organization. There can be significant legal and financial penalties if a business process leaks sensitive information. For example, a company which leaks trade secrets exposes itself to unwanted competition, and a company which leaks customer information may have to deal with angry customers.

When workflows are carried out manually, it is necessary to rely on the honesty of the individuals involved to obey organizational policies regarding the appropriate use and sharing of sensitive information. This is practical for workflows on a small scale, where there are legal mechanisms such as contracts designed to protect information. However workflows are increasingly becoming automated, in part or in full. The automated portions of these workflows may be too complex or process too much information for humans to take responsibility for their correct operation on a daily basis, and it is unclear who is legally responsible if an automated process malfunctions. Therefore it is important to verify that automated workflows obey organizational policies regarding the sharing of sensitive information.

We propose using the Orc concurrency calculus as a language for encoding workflows. By verifying the information flow security of an Orc program, we indirectly verify the security of the workflow it describes. The remainder of the paper is organized as follows. In Section 2, we describe the Orc

$$\begin{array}{rllll}
\text{variable name} & x & & & \\
\text{callable name} & M & & & \\
\text{Orc program} & & := \{D, \cdots\} \quad E & & \\
\text{function declaration} & D & := \texttt{def}\ \ M(x, \cdots) = E & & \\
\text{Orc expression} & E & := & & \\
\text{site call} & & & & M(x, \cdots) \\
\text{parallel} & & & \mid & E\ \ \texttt{|}\ \ E \\
\text{sequential} & & & \mid & E\ \ \texttt{>}x\texttt{>}\ \ E \\
\text{pruning} & & & \mid & E\ \ \texttt{<}x\texttt{<}\ \ E \\
\text{otherwise} & & & \mid & E\ \ \texttt{;}\ \ E
\end{array}$$

Table 1: Orc calculus syntax

calculus and argue that it has advantages over other concurrency calculi for describing and analyzing workflows. In Section 3, we describe informally what it means for an Orc program to be secure. In Section 4, we sketch a type system for verifying the security of Orc programs. In Section 5, we discuss related work, and conclude with a summary of our contributions in Section 6.

2 Orc calculus

The Orc concurrency calculus, originally conceived as a language for distributed and concurrent computing [10], is capable of straightforwardly describing common workflow patterns [2].

The primary advantage of Orc over typical workflow description languages such as BPML [4] is that Orc has a simple formal semantics [7]. This is possible because the Orc language only has a few primitive concepts which can be combined to model arbitrary complex behaviors.

Orc's primary advantage over other concurrency calculi such as CSP [5] or the π calculus [9] is that the lexical structure of an Orc program constrains the concurrency and communication structure of its execution. As long as a program uses only the Orc combinators for communication, it is trivial to prove properties such as freedom from deadlock. Orc programs can also invoke external services, called *sites*, which provide arbitrary computation and communication services. Because Orc makes few assumptions about the behavior of sites, they can be used to model essential features like channels, mutable memory, and network communication, as well as less common features like web service invocation and interaction with human agents. By distinguishing structured concurrency primitives (combinators) from unstructured external services (sites), Orc is easy to analyze in the common case, but flexible enough to encode arbitrary workflows, without having to shoehorn all communication into a single primitive such as channels.

In the remainder of this section, we give a brief summary of the version of the Orc calculus used in this paper. For a more complete description of the calculus, see Kitchin et al [8].

The syntax of the Orc calculus is given in Table 1. We use an ellipsis ($\cdots$) to indicate that zero or more of the immediately preceding form is allowed by the syntax. An Orc program consists of one or more function declarations followed by a goal expression.

The semantics of the Orc calculus is based on the *execution* of expressions. When executed, an expression may invoke *sites* and *publish* zero or more values. When an execution is complete, we say that it has *halted*.

2.1 Site calls

The simplest Orc expression is a site call of the form $M(x, \cdots)$. Site calls model the invocation of external services, called *sites*. Orc assumes nothing about the internal behavior of sites: they may be stateful or functional, isolated or communicating, local or remote, reliable or unreliable. When executed, a site call sends the values of its arguments to the site and waits for a single response. If the site responds with a value, the site call publishes that value. If the site responds that it has no value to return, the site call halts without publishing a value. If the site fails to respond, the site call waits forever and does not halt.

Though the Orc calculus itself defines no sites, there are a few fundamental sites which are so essential to writing useful computations that we always assume they are available. The site `let` is the identity site; the call $\texttt{let}(x)$ responds with the value of x. The site `if` is used for conditional control flow; the call $\texttt{if}(x)$ responds with a signal (a value which carries no information) if x has the value `True`, and otherwise halts. The site `Rtimer` is used to model the passage of time; the call $\texttt{Rtimer}(x)$ responds with a signal after x milliseconds have elapsed.

2.2 Combinators

The Orc calculus uses four primitive combinators to structure concurrency and communication within an expression. Combinators compose two Orc expressions to form a larger combined expression. In general, the execution of a combined expression halts when the executions of its subexpressions have halted.

Parallel The parallel combinator models simple parallelism without communication. A parallel expression has the form $E \texttt{ | } F$. When executed, this expression executes both E and F in parallel and publishes the union of their publications.

Sequential The sequential combinator models sequencing and pipeline communication. A sequential expression has the form $E \texttt{ >}x\texttt{> } F$. When executed, this expression first executes E. For each publication of E, F is executed with the variable x bound to the value of the publication. The sequential expression publishes the union of the values published by executions of F. When x is not used in F, this form may be abbreviated $E \texttt{ >> } F$.

Pruning The pruning combinator models one-way data-flow communication and forced termination. A pruning expression has the form $F \texttt{ <}x\texttt{< } E$. When executed, this expression begins by executing both F and E in parallel. E will provide a value for the data-flow variable x which is bound in F. If any site call in F attempts to read the value of x before it is available, execution of that site call is suspended until the value becomes available. As soon as E publishes a value, that value is assigned to x and execution of E is immediately terminated. The pruning expression publishes the values published by F. When x is not used in F, this form may be abbreviated $F \texttt{ << } E$.

Otherwise The otherwise combinator models termination detection and error recovery. An otherwise expression has the form $E \texttt{ ; } F$. When executed, this expression first executes E, publishing the values published by E. If E halts without publishing any value, then F is executed, and the otherwise expression publishes the values published by F.

2.3 Function definitions

The Orc calculus includes functions, which provide abstraction and recursion. A function declaration of the form $\texttt{def}\ M(x, \cdots) = E$ introduces a function with the name M, formal parameters $x, \cdots$, and body E. Function names are scoped over the entire Orc program, so functions can call themselves or other functions recursively.

Functions are called using the site call syntax $M(x, \cdots)$, where M is a function name and $x, \cdots$ are the arguments to the call. When a function is called, its body is executed with the formal parameters given in the definition replaced by the arguments to the call. The function call publishes the values published by the body of the function. This call-by-name strategy means that function calls, unlike site calls, can begin executing before the values of their arguments are available.

In this presentation of the Orc calculus, function names are syntactically distinguished from variable names, so functions cannot be used as first-class values and higher-order functions cannot be defined. This limitation is not fundamental (Kitchin et al [8] describe an implementation of Orc which includes first-class functions), but it simplifies analysis.

3 Orc information flow security

In this section, we explore what it means informally for an Orc program to be secure. In the broadest terms, a secure Orc program should only share information with parties who have been authorized to access it.

3.1 Channels

In the security literature, any mechanism which allows the transmission of information is called a *channel*. Channels can be either *direct*, when they are designed to convey information, or *covert*, when the transmission of information is not their intended purpose.

Direct channels Examples of direct channels include shared memory, the file system, inter-process communication, and network sockets. In Orc, all direct channels are modeled by sites. This is in contrast to languages which model direct channels as mutable memory locations, such as the sequential language analyzed by Denning and Denning [3]. While the two approaches are formally equivalent (a site call can be modeled by writing to one memory location followed by reading from another), modeling site calls directly has some advantages. When we pass information to a specific site, the identity of the site may tell us something about how that information will be used. This knowledge can be used to improve our security analysis. When we write to a memory location, we don't know exactly how the information at that location may be used, so our security analysis has to be more conservative. The distinction will become clearer when we discuss the security properties of sites in the next section.

Covert channels Examples of covert channels include program control flow, non-termination, timing, scheduler behavior, cache behavior, and resource consumption. Since our goal is to analyze workflows described in Orc, independent of any particular implementation, we will not consider implementation-specific channels such as cache behavior. This leaves program control flow, non-termination, and timing as the primary covert channels we are concerned with. As we shall see, the most important property of these covert channels is that they mask the source of information, but they still need a direct channel to transmit it. Therefore, in order to verify the security of a program, it is only necessary to check that direct channels are used appropriately.

Control flow In Orc, program control flow and non-termination are intimately related. Orc does not have explicit control flow statements such as `if` ... `then` ... `else`. Instead, conditional branching is modeled by executing both branches in parallel and expecting the untaken branch to halt before performing useful work. For example, the following program calls the site `M`, prints `"True"` if `M()` responds `True`, and prints `"False"` otherwise:

```
M()      >ok>
not(ok) >nok>
(
     if(ok)   >> print("True")
   | if(nok) >> print("False")
)
```

In this program, both branches of the conditional begin executing, but only one of the calls to `if` will return, so only one call to `print` will execute. This example also illustrates how control flow can be used as a covert channel for transmitting information. Although the program never directly transmits the value of `M()` to the `print` site, it is clear that the `print` site can infer the value of `M()`. If `M()` returns sensitive information and the output of `print` is publicly accessible, then this program is insecure.

The `if` site introduces a covert channel because it encodes information about its arguments in the number of values it publishes (zero or one). In general, any expression which may fail to publish can use this fact to leak information. In this respect, expressions which halt without publishing and expressions which neither publish nor halt are equivalent. Any security analysis which aims to identify these covert channels must account for the general (non-)termination properties of expressions.

Timing In concurrent programs, the timing behavior of processes can be used to transmit information. Consider the following program (the site `write` updates the value of a memory cell which is read by `read`):

```
write(false) >>
(
     Rtimer(n) >> write(true)
   | Rtimer(100) >> read() >x> M(x)
)
```

This program tells the site `M` whether the value of some variable `n` is less than 100. If `n` < 100, then the value read from the memory cell and sent to `M` will be True, otherwise it will be False. If `n` is sensitive information and `M` is an insecure site, then this program is insecure.

Zdancewic and Myers [15] propose a distinction between *internal* and *external* timing channels. Internal timing channels use data races to transmit information between different threads in a single program. External timing channels require that some third party be able to monitor the execution time of processes or the program as a whole without any explicit action by the program. In our analysis, we consider only internal timing channels.

3.2 Site security properties

Since all communication between Orc and its environment is mediated by sites, understanding how sites use information is critical to proving the security of Orc programs. Sites can both send information to and receive information from third parties. These third parties may be other sites being used by an Orc program or external entities. Since we wish to restrict the flow of information in a program, the most relevant properties of a site are what information it shares and who it shares that information with.

An Orc program can send the following information to third parties via a site call:

- When/whether the site was called
- The value of each argument

A site call is secure only if it satisfies pre-conditions related to the third parties with whom the site communicates. For example, if a site is called under certain conditions depending on sensitive information, it cannot share the fact that it was called with third parties not authorized to access that information. There is one further pre-condition connected to the availability of the arguments to the site call. Recall that a site call cannot proceed unless all its arguments are available. Therefore any party who knows that a site was called can infer that all the arguments to the call were available. If the availability of any of the arguments depends on sensitive information, then the site cannot share the fact that it was called with third parties not authorized to access that information. Beyond this, there is no further information which an Orc program can share through a site call.

An Orc program can receive the following information from third parties via a site call:

- When/whether the site returned
- The value returned

A site call is secure only if the Orc program satisfies post-conditions on how it uses the information returned by the call. For example, if a site returns under certain conditions depending on sensitive information, the Orc program cannot share the fact that the site returned with third parties not authorized to access that information. Because Orc can only share information through site calls, this means that the post-conditions of a site call must satisfy the pre-conditions of the site calls which depend on it.

In the next section, we will see how these pre- and post-conditions can be formally described and checked using security types.

4 Secure type system for Orc

In the previous section we discussed criteria which should be used to decide whether workflows expressed as Orc programs are secure. Now we address the question of how these criteria may be implemented. Broadly, there are two main approaches to ensuring that security policies are followed: dynamic enforcement, and static verification.

Dynamic enforcement means that whenever a program attempts to communicate information to a third party, the runtime must first check that this communication is authorized based on the information and third party involved. The HiStar operating system [16] exemplifies this approach. The main advantage of dynamic enforcement is that little or no support from programming languages is necessary, as long as all communication goes through a secure kernel. The main disadvantage of dynamic enforcement, from the point of view of workflow security, is that once a workflow has started running, it is too late to find out whether it is secure or not. Unlike typical concurrent programs, workflows run on a human timescale and may take months to complete. Workflows may also coordinate activities between multiple distributed parties. If a workflow encounters a security error long after it has been started, it may not be possible to checkpoint the workflow's state, repair it, and restart it.

Therefore we believe that static verification is a better approach for ensuring workflow security. In the remainder of this section, we sketch a type system for statically verifying the information flow security properties of workflows expressed in the Orc calculus. We argue informally that this type system encodes the relevant security properties. A formal proof of subject reduction and non-interference of well-typed programs remains a topic for future work.

4.1 Security labels

In the tradition of Denning and Denning [3], we assume that every piece of information (runtime value) has an associated security label which signifies the minimum clearance necessary to access it. Security labels form a lattice, meaning that there is a partial order $\sqsubseteq$ over labels and any two labels

security label	L	
variable name	x	
callable type	T	$:= (x[L] \Leftarrow x[L], \cdots)[L] \rightarrow \exists x[L], \cdots .\{E \Leftarrow E, \cdots\}$
expression template	E	$:= \mathsf{True} \mid \mathsf{False} \mid \mathsf{signal} \mid E \vee E \mid E \wedge E \mid \neg E \mid x$
callable name	M	
callable type assertion		$:= M : T$
Orc expression	O	
function declaration		$:= \texttt{def}\ \ M(x, \cdots) : T = O$
secure type judgement		$:= \Gamma, L \vdash O : \{F \Leftarrow F, \cdots\}$
variable context	Γ	$:= \langle x \mapsto (F \Leftarrow F), \cdots \rangle$
expression type	F	$:= \mathsf{True} \mid \mathsf{False} \mid \mathsf{signal} \mid F \vee F \mid F \wedge F \mid \neg F \mid x[L]$

Table 2: Secure type syntax

L_1 and L_2 have a least upper bound (or join) $L_1 \sqcup L_2$. We will assume a minimal security label $\bot$, which labels unrestricted information, and a maximal security label $\top$, which labels the most restricted information.

Security labels may be viewed as sets of principals (persons or entities who may try to access sensitive information), where $L = \{p, \ldots\}$ means that each principle p is *not allowed* to access information labeled by L. In this view, the partial order $\sqsubseteq$ is implemented by $\subseteq$, and the join operation $\sqcup$ is implemented by $\cup$. The minimal label $\bot = \emptyset$, and the maximal label $\top = D$ where D is the domain of principals.

4.2 Secure type syntax

The syntax used by the secure type system is given in Table 2. We use an ellipsis $(\cdots)$ to indicate that zero or more of the immediately preceding form is allowed by the syntax.

4.2.1 Callable types

Callable types T are used to describe both functions and sites, since the possible behaviors of sites form a subset of the possible behaviors of functions.

Argument list The first part of the callable type is the argument list: $(x_v[L_v] \Leftarrow x_p[L_p], \cdots)$. Each item $x_v[L_v] \Leftarrow x_p[L_p]$, read "$x_v$ (labeled L_v) when x_p (labeled L_p)", represents a single argument to the callable, where x_v binds to the value of that argument and x_p binds to the conditions under which that argument is available. The label L_v places an upper bound on the security label of the corresponding argument's value; the callable asserts that it will not share the value of the argument with any entity having security clearance $\sqsubset L_v$. Similarly, the label L_p places an upper bound on the security label of the corresponding argument's publication conditions; the callable asserts that it will not share the availability of the argument with any entity having security clearance $\sqsubset L_p$. It is always the case that $L_p \sqsubseteq L_v$, since an argument value cannot be shared without also sharing the fact that the argument is available. In the case of sites, the L_ps must all be equal, since a site is only called if all arguments are available; knowing that one argument is available implies the others are as well. As an abbreviation, any label $[\top]$ in the argument list may be elided, so $x_v \Leftarrow x_p$ is equivalent to $x_v[\top] \Leftarrow x_p[\top]$.

Security context Immediately following the argument list is the security context $[L]$. This places an upper bound on the conditions under which the callable may be called; the callable asserts that it will not share the fact that it was called with any entity having security clearance $\sqsubset L$. It is always the case that $L \sqsubseteq L_p \sqsubseteq L_v$, since a callable cannot share information about its arguments without revealing that it was called. In the case of sites, each $L_p = L$, since the fact that the site was called implies that all its arguments are available. As an abbreviation, the security context $[\top]$ may be elided.

Publication template To the right of → is the publication template. The publication template begins with existential variable declarations $\exists x[L], \cdots$. Each existential variable declaration introduces a variable bound to an unknown expression of the given security level. The order of variables in the declarations is not significant. The applications of existential variables will become clear when we give examples of site type declarations in the next section. As an abbreviation, the $\exists.$ may be elided when there are no existential variables, so $\{E_v \Leftarrow E_p, \cdots\}$ is equivalent to $\exists.\{E_v \Leftarrow E_p, \cdots\}$.

Following the existential variable declarations is a set representing possible publications. Each publication $E_v \Leftarrow E_p$, read "E_v when E_p", represents a value, E_v, and the conditions under which it is published, E_p. Both the value and its publication conditions may be complex expression templates. Each variable in these expression templates refers either to an argument bound by the callable's argument list, or to the unknown expression bound by an existential declaration. Expression templates may include the constant `signal`, representing a signal value which contains no information.

Limitations The syntax imposes two important limitations on the secure type system for the sake of simplicity. First, since each security label in the type must be named concretely, it is not possible to parameterize the security labels of existentially-quantified variables in the publications by the security labels of the arguments. For example, the type of a site like `add`, which publishes a value whose label depends on the labels of the inputs, cannot by given in a single declaration; instead one declaration must be given for every possible combination of input security labels. This limitation is not fundamental and could be easily corrected, at the expense of slightly more verbose typing rules.

The second limitation is that callable names (sites and functions) are distinct from program variables, so higher-order functions are not possible. In order to support such a feature, it would be necessary to track not only the label but also the type (boolean, callable, or other) of each program value. Since this would significantly complicate the presentation without fundamentally changing the nature of the secure information flow verification, we have omitted this feature for simplicity.

4.2.2 Secure type judgements

The security of an Orc program is asserted by a secure type judgement together with a set of site and function type declarations.

Callable type assertions simply assert the callable type (security properties) of a callable used by the program. For sites, we assume these assertions are correct; it is the responsibility of the programmer to verify the correctness of each site's type relative to its implementation.

Function declarations give the callable type of a function together with the function's implementation. With this information we can derive a callable type assertion for the function and verify that the function's implementation conforms to its type, i.e. it does not violate the security properties asserted by the type.

Secure type judgements have the form $\Gamma, L \vdash O : \{F \Leftarrow F, \cdots\}$. This judgement asserts that the Orc expression O has the set of publications $\{F \Leftarrow F, \cdots\}$, given variable context Γ and security context L. Each publication $F_v \Leftarrow F_p$, read "F_v when F_p", represents a value F_v published under the condition that F_p holds. F_v and F_p are abstract expressions consisting of complex boolean expressions, signals, and labeled variables $x[L]$. Each labeled variable is implicitly existentially

$$
\begin{aligned}
\mathtt{let} &: (x_v \Leftarrow x_p) \rightarrow \{x_v \Leftarrow x_p\} \\
\mathtt{if} &: (x_v \Leftarrow x_p) \rightarrow \{\mathsf{signal} \Leftarrow x_v \wedge x_p\} \\
\mathtt{not} &: (x_v \Leftarrow x_p) \rightarrow \{\neg x_v \Leftarrow x_p\} \\
\mathtt{or} &: (x_v \Leftarrow x_p, y_v \Leftarrow y_p) \rightarrow \{x_v \vee y_v \Leftarrow x_p \wedge y_p\} \\
\mathtt{plus} &: (x_v[L_x] \Leftarrow x_p, y_v[L_y] \Leftarrow y_p) \rightarrow \exists z[L_x \sqcup L_y].\{z \Leftarrow x_p \wedge y_p\} \\
\\
\mathtt{writeL} &: (x_v[\bot] \Leftarrow x_p[\bot])[\bot] \rightarrow \{\mathsf{signal} \Leftarrow x_p\} \\
\mathtt{readL} &: () \rightarrow \exists y_v[\bot].\{y_v \Leftarrow \mathsf{True}\} \\
\mathtt{writeH} &: (x_v \Leftarrow x_p) \rightarrow \{\mathsf{signal} \Leftarrow x_p\} \\
\mathtt{readH} &: () \rightarrow \exists y_v[\top].\{y_v \Leftarrow \mathsf{True}\} \\
\\
\mathtt{putL} &: (x_v[\bot] \Leftarrow x_p[\bot])[\bot] \rightarrow \{\mathsf{signal} \Leftarrow x_p\} \\
\mathtt{getL} &: ()[\bot] \rightarrow \exists y_v[\bot], y_p[\bot].\{y_v \Leftarrow y_p\} \\
\mathtt{putH} &: (x_v \Leftarrow x_p) \rightarrow \{\mathsf{signal} \Leftarrow x_p\} \\
\mathtt{getH} &: () \rightarrow \exists y_v[\top], y_p[\top].\{y_v \Leftarrow y_p\} \\
\\
\mathtt{Rtimer}_1 &: (x_v \Leftarrow x_p) \rightarrow \exists y_p[\top].\{\mathsf{signal} \Leftarrow y_p \wedge x_p\} \\
\mathtt{Rtimer}_2 &: (x_v \Leftarrow x_p[\bot])[\bot] \rightarrow \exists y_p[\bot].\{\mathsf{signal} \Leftarrow y_p \wedge x_p\}
\end{aligned}
$$

Table 3: Example site type assertions

quantified (with scope over the entire proof) and represents some unknown value of the given security level.

The variable context Γ maps each program variable to a publication $(F_v \Leftarrow F_p)$, indicating that the variable has the value F_v if F_p holds. The security context L represents the level of information which could be inferred by the fact that the Orc expression O is being executed. For example, in the program `if(topSecret) >> M()`, if the expression `M()` executes then we can infer that `topSecret`=True, so the security context of `M()` must be at least as high as the security label of `topSecret`'s value.

4.3 Site types

We give some examples of type assertions for representative Orc sites in Table 3. Because Orc sites are strict and publish no more than one value, site types always have the form $(x_v[L_v] \Leftarrow x_p[L], \cdots)[L] \rightarrow \exists y[L_y], \cdots.\{E_v \Leftarrow E_p \wedge (x_p \wedge \cdots)\}$.

The first several sites are straightforward. The type of the site `let` asserts that it accepts one argument and publishes that argument's value under the condition that it is available. The type of the site `if` asserts that it accepts one argument and publishes a signal contingent on the fact that the argument was true and available. The sites `not` and `or` assert that they perform the expected operation on their argument's value.

The site `plus` returns the sum of its arguments, and provides a good example of functional binary operators in general. Since information about either argument may be inferred from the return value, the label of the return value is the least upper bound of the labels of the arguments. This is not syntactically a site type, but rather represents a family of types, parameterized by the security labels

$$\begin{aligned}
\text{label}(\mathsf{True}) &= \bot \\
\text{label}(\mathsf{False}) &= \bot \\
\text{label}(E \vee F) &= \text{label}(E) \sqcup \text{label}(F) \\
\text{label}(E \wedge F) &= \text{label}(E) \sqcup \text{label}(F) \\
\text{label}(\neg E) &= \text{label}(E) \\
\text{label}(x[L]) &= L
\end{aligned}$$

Table 4: Expression labels

of the arguments. In order to actually analyze a site like `plus`, it would be necessary to annotate each application of `plus` with information that would allow us to identify the appropriate type (or this information could be inferred automatically).

The sites `writeL` and `readL` write to and read from a low-security memory cell. Only low-security information may be written to a low-security cell, and values read from the cell always have a low-security label. Since any call `writeL` may be observed in a low-security context by `readL`, `writeL` can only be called in a low-security context. The high-security equivalents are `writeH` and `readH`. Note that any information in any security context may be written to a high-security cell, but values read from the cell always have a high-security label.

The sites `putL` and `getL` write to and read from a low-security channel. This is an asynchronous blocking channel, so `putL` always succeeds but `getL` blocks if no value is available. The types of these sites are very similar to those of `writeL` and `readL`, with one important difference. Since `getL` removes an item from the channel, it may cause the channel to become empty and block some other call to `getL`. This would communicate information, so `getL` can only be called in a low-security context. Furthermore, `getL` may or may not return depending on whether `putL` was called, which is expressed by the fact that `getL`'s publication conditions depend on some unknown expression. The high-security equivalents are `putH` and `getH`.

The site `Rtimer` poses an interesting problem. Without a formal semantics of time, we must conservatively assume that every call to `Rtimer` introduces a potential causal dependence (and therefore communication) with every other possible call to `Rtimer`. There are two possible ways to express this fact in our type system. $\texttt{Rtimer}_1$ asserts that `Rtimer` may be called in any context (secure or insecure), but its return is potentially contingent upon secure information (from another call to `Rtimer` in a secure context). $\texttt{Rtimer}_2$, in contrast, asserts that `Rtimer` may only be called in an insecure context, so its return is guaranteed to be contingent only upon insecure information. Other variations are possible; for example, allowing calls to `Rtimer` in some intermediate security context. In any case, only one of these site types should be used in any particular program.

4.4 Secure type semantics

The semantics of a secure type judgement are given by inference rules which allow the judgement to be proven. These inference rules constitute a formal realization of our intuitions about the properties a secure program should have.

4.4.1 Expression labels

In order to find the security level of a variable or publication, it is necessary to infer the label of that variable or publication's corresponding expression value. The label of an expression is calculated as defined in Table 4. These rules are straightforward: constants are public, operations are labeled with the join of their operand labels, and variables are labeled as given. Before finding the label

$$\frac{\begin{array}{c}\{E'_v \Leftarrow E'_p, \cdots\} = \{E_v \Leftarrow E_p, \cdots\}[F_y/y, \cdots] \\ \text{label}(F_y) \sqsubseteq L_y \quad \cdots\end{array}}{\{E'_v \Leftarrow E'_p, \cdots\} \text{ instantiates } \exists y[L_y], \cdots.\{E_v \Leftarrow E_p, \cdots\}} \ (\textsc{Instantiates})$$

$$\frac{}{\{E_v \Leftarrow \mathsf{False}, F_v \Leftarrow F_p, \cdots\} = \{F_v \Leftarrow F_p, \cdots\}} \ (\textsc{False-Elim})$$

$$\frac{}{\{E_v \Leftarrow E_p, E_v \Leftarrow E'_p, F_v \Leftarrow F_p, \cdots\} = \{E_v \Leftarrow E_p \vee E'_p, F_v \Leftarrow F_p, \cdots\}} \ (\textsc{Merge})$$

Table 5: Manipulating publication sets

of an expression, it should be simplified as much as possible using standard boolean identities to eliminate subexpressions which do not affect the overall value.

These rules do not account for the *amount* of information leaked by an expression. For example, an expression such as `password = "letmein"` can only leak one bit of information about `password`: whether it is equal to `"letmein"` or not. In some cases, such as a password authorization program, leaking this small bit of information may be acceptable and even necessary. However our approach will conservatively classify an expression as high-security if there is a chance that it contains even one bit of high-security information. This limitation is typical of many approaches to verifying information flow security; the underlying problem and possible solutions are discussed by Zdancewic [14].

4.4.2 Publication sets

In several places, the secure type system needs to manipulate publication sets which abstractly represent the possible publications of an expression. For example, the publication set of a parallel expression is constructed from the publication sets of its constituents. The most important rules governing publication sets are given in Table 5.

The first rule, Instantiates, states that a publication set *instantiates* an existentially-quantified template if there exists some substitution of the existentially-quantified variables (satisfying constraints on the variables' labels) which makes the template equal to the publication set. Intuitively, the publication set must be a *witness* to the existentially-quantified template.

The second rule, False-Elim, allows publications which are never available to be removed from a publication set. This rule reflects the fact that a publication set represents a set of *possible* publications. Therefore a value which doesn't appear in the set is equivalent to a value which appears in the set but is published under no condition.

The final rule, Merge, allows publication conditions for the same publication value to be combined. Because each item in a publication set represents a possible publication, if the same value appears twice with different conditions, this is equivalent to the value appearing once under the disjunction of those conditions. This rule is necessary to expose opportunities to simplify publication conditions when computing labels or comparing publication sets.

In addition to the non-standard rules described above, we assume standard set identities such as equality under reordering and duplication of elements, as well as standard boolean identities for the expressions within a set.

4.4.3 Secure typing rules

The complete rules for our proposed secure type system are given in Table 6. In these rules we use ellipses $(\cdots)$ in an intuitive but rigorous manner, inspired by the Scheme macro system [6]. When an ellipsis appears in a binding context, variables in the form preceding the ellipsis range over zero or

$$\frac{\Gamma, L \vdash F : \{F_v \Leftarrow F_p, \cdots\} \qquad \Gamma, L \vdash G : \{G_v \Leftarrow G_p, \cdots\}}{\Gamma, L \vdash F \mid G : \{F_v \Leftarrow F_p, \cdots\} \cup \{G_v \Leftarrow G_p, \cdots\}}\ \text{(Parallel)}$$

$$\frac{\begin{array}{c}\Gamma, L \vdash F : \{F_v \Leftarrow F_p, \cdots\} \\ \Gamma[x \mapsto (F_v \Leftarrow F_p)], L \sqcup \text{label}(F_p \vee \cdots) \vdash G : \{G_v \Leftarrow G_p, \cdots\} \quad \cdots\end{array}}{\Gamma, L \vdash F \mathbin{\texttt{>}x\texttt{>}} G : \{G_v \Leftarrow F_p \wedge G_p, \cdots\} \cup \cdots}\ \text{(Sequential)}$$

$$\frac{\begin{array}{c}\Gamma, L \vdash F : \{F_v \Leftarrow F_p, \cdots\} \\ \Gamma[x \mapsto (\mathsf{signal} \Leftarrow \mathsf{False})], L \vdash G : \{G'_v \Leftarrow G'_p, \cdots\} \\ \Gamma[x \mapsto (F_v \Leftarrow F_p)], L \vdash G : \{G_v \Leftarrow G_p, \cdots\} \quad \cdots\end{array}}{\Gamma, L \vdash G \mathbin{\texttt{<}x\texttt{<}} F : \{G'_v \Leftarrow G'_p\} \cup (\{G_v \Leftarrow F_p \wedge G_p, \cdots\} \cup \cdots)}\ \text{(Pruning)}$$

$$\frac{\begin{array}{c}\Gamma, L \vdash F : \{F_v \Leftarrow F_p, \cdots\} \\ F'_p = \neg(F_p \vee \cdots) \\ \Gamma, \text{label}(F'_p) \sqcup L \vdash G : \{G_v \Leftarrow G_p, \cdots\}\end{array}}{\Gamma, L \vdash F \ ;\ G : \{F_v \Leftarrow F_p, \cdots\} \cup \{G_v \Leftarrow F'_p \wedge G_p, \cdots\}}\ \text{(Otherwise)}$$

$$\frac{\begin{array}{c}M : (x_v[L_v] \Leftarrow x_p[L_p], \cdots)[L'] \to \exists y[L_y], \cdots.\{E_v \Leftarrow E_p, \cdots\} \\ L \sqsubseteq L' \\ \Gamma(x) = (F_v \Leftarrow F_p) \quad \cdots \\ \text{label}(F_v) \sqsubseteq L_v \quad \cdots \\ \text{label}(F_p) \sqsubseteq L_p \quad \cdots \\ y' \text{ fresh} \quad \cdots\end{array}}{\Gamma, L \vdash M(x, \cdots) : \{E_v \Leftarrow E_p, \cdots\}[y'[L_y]/y, \cdots][F_v/x_v, F_p/x_p, \cdots]}\ \text{(Call)}$$

$$\frac{\begin{array}{c}\texttt{def}\ M(x, \cdots) : (x_v[L_v] \Leftarrow x_p[L_p], \cdots)[L] \to \exists y[L_y], \cdots.\{E_v \Leftarrow E_p, \cdots\} = F \\ \Gamma[x \mapsto (x_v[L_v] \Leftarrow x_p[L_p]), \cdots], L \vdash F : \{E'_v \Leftarrow E'_p, \cdots\} \\ \{E'_v \Leftarrow E'_p, \cdots\} \text{ instantiates } \exists y[L_y], \cdots.\{E_v \Leftarrow E_p, \cdots\}[x_v[L_v]/x_v, x_p[L_p]/x_p, \cdots]\end{array}}{M : (x_v[L_v] \Leftarrow x_p[L_p], \cdots)[L] \to \exists y[L_y], \cdots.\{E_v \Leftarrow E_p, \cdots\}}\ \text{(Def)}$$

Table 6: Secure typing rules

more instances of that form. When an ellipsis appears in a non-binding context, the form preceding the ellipsis is instantiated once for each instance of its variables. We also assume for clarity some standard syntactic sugar for variable contexts Γ:

$\Gamma(x) =$ the left-most E such that $\Gamma = \langle \ldots, x \mapsto E, \ldots \rangle$.

$\Gamma[x \mapsto E] = \langle x \mapsto E, x' \mapsto E', \cdots \rangle$ where $\Gamma = \langle x' \mapsto E', \cdots \rangle$.

The Parallel rule checks expressions of the form $F \mid G$. It simply asserts that the publication set of $F \mid G$ is the union of the publication sets of F and G. The Merge rule discussed in the previous section may be used here to combine publications from both branches.

The Sequential rule checks expressions of the form $F \mathbin{\texttt{>}x\texttt{>}} G$. We are given that F publishes some set of publications. For each of these publications, we bind x to the publication and find the publications of G in this context. The security context for G is the level of information which can be inferred simply from knowing that F published, i.e. the join of the current security context with the label of the disjunction of all possible publication conditions of F. Finally, the overall expression publishes the union of all the possible publications of G, under the condition that F published.

The Pruning rule checks expressions of the form $G \mathbin{\texttt{<}x\texttt{<}} F$. Since the pruning combinator executes both sides regardless of whether F publishes any values, we first find the publications of G assuming that F does not publish anything (binding x to a meaningless value which is never available). Next, we assume that F does publish something, and we find the publications of G for each of these possible publications. Note that the security context of G is not directly affected by F; it will only need to be raised if and when G actually checks for the availability of x. Finally, the overall expression publishes the union of the possible publications of G.

The OTHERWISE rule checks expressions of the form F ; G. Similarly to PARALLEL, such an expression publishes the union of possible publications of F and G. However G only executes if F does not publish, affecting both the security context of G and the publication conditions of G.

The CALL rule checks that a call of the form $M(x, \cdots)$ conforms to the security properties expected by M. Given a type assertion for M, the rule first checks that the security context is valid. Then the rule looks up each argument to the call in the variable environment and checks that the argument's actual labels conform to the labels required by M. Finally, the publication set of the call is constructed from the template provided by M, generating unique names for each existentially-quantified variable in the template and substituting argument expressions for argument variables in the template.

The DEF rule verifies a function type declaration by comparing its declared publications to its actual publications. Given a function type declaration, the rule first finds the publications of the function's body, given the variable context asserted by the argument types. The rule then checks that the resulting publication set is an instance of the declared publication set template. Argument variables in the template are replaced by their (abstract) values, while existential variables in the template are handled by the INSTANTIATES rule from the previous section. If the publication sets match, then the function satisfies the asserted security properties, and the derived type assertion is valid.

4.5 Discussion

We will examine two trivial examples to illustrate how the secure type system applies to complete programs. In these examples, we will make use of the `readH`, `writeH`, `readL`, and `writeL` sites defined in Section 4.3.

The first example program P reads a value from a high-security memory cell, inverts it (in a contrived manner), and then writes a value to a low-security memory cell. By inspection, this program is secure.

```
readH() >h>
not(h) >nh>
(   if(h)  >> writeH(false)
  | if(nh) >> writeH(true)
) >>
writeL(true)
```

Here are the essential parts of the type derivation showing that P is secure:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, \bot \vdash \texttt{if(h) ...} : \{\mathsf{signal} \Leftarrow h[\top]\}} \quad \dfrac{\vdots}{\Gamma, \bot \vdash \texttt{if(nh) ...} : \{\mathsf{signal} \Leftarrow \neg h[\top]\}}}{\Gamma, \bot \vdash \texttt{(if(h) ... | if(nh) ...)} : \{\mathsf{signal} \Leftarrow h[\top], \mathsf{signal} \Leftarrow \neg h[\top]\}}}{\Gamma, \bot \vdash \texttt{(if(h) ... | if(nh) ...)} : \{\mathsf{signal} \Leftarrow \mathsf{True}\}}}{}$$

$$\vdots$$

$$\dfrac{\Gamma = \langle \mathtt{h} \mapsto (h[\top] \Leftarrow \mathsf{True}), \mathtt{nh} \mapsto (\neg h[\top] \Leftarrow \mathsf{True})\rangle \quad \dfrac{\vdots}{\Gamma, \bot \vdash \texttt{writeL(true)} : \{\mathsf{signal} \Leftarrow \mathsf{True}\}}}{\langle\rangle, \bot \vdash P : \{\mathsf{signal} \Leftarrow \mathsf{True}\}}$$

The key step in this derivation is the merging of the publication sets of the two `if` branches. This step proves that the parallel expression always publishes, so the subsequent `writeL` can execute in a low-security context.

The second example program P' is the same as P with one seemingly minor change: if `h` is true, execution only proceeds if the low-security value `l` is also true. This program is insecure but this may not be obvious by inspection.

```
readL() >l>
readH() >h>
not(h) >nh>
(   if(h)  >> if(l) >> writeH(true)
  | if(nh) >> writeH(false)
) >>
writeL(true)
```

Here are the essential parts of the type derivation showing that P' is insecure:

$$\frac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, \bot \vdash \texttt{if(h) >> if(l) } \ldots : \{\mathsf{signal} \Leftarrow h[\top]\}} \qquad \dfrac{\vdots}{\Gamma, \bot \vdash \texttt{if(nh) } \ldots : \{\mathsf{signal} \Leftarrow \neg h[\top]\}}}{\Gamma, \bot \vdash \texttt{(if(h) >> if(l) } \ldots \texttt{ | if(nh) } \ldots \texttt{)} : \{\mathsf{signal} \Leftarrow (h[\top] \wedge l[\bot]), \mathsf{signal} \Leftarrow \neg h[\top]\}}}{\Gamma, \bot \vdash \texttt{(if(h) >> if(l) } \ldots \texttt{ | if(nh) } \ldots \texttt{)} : \{\mathsf{signal} \Leftarrow l[\bot] \vee \neg h[\top]\}}}{\vdots}$$

$$\frac{\Gamma = \langle \texttt{l} \mapsto (l[\bot] \Leftarrow \mathsf{True}), \ldots \rangle \qquad \mathrm{label}(l[\bot] \vee \neg h[\top]) = \top \qquad \dfrac{\begin{array}{c} \texttt{writeL} : (\ldots)[\bot] \rightarrow \{\ldots\} \\ \top \not\sqsubseteq \bot \end{array}}{\Gamma, \top \nvdash \texttt{writeL(true)} : \{\mathsf{signal} \Leftarrow \mathsf{True}\}}}{\langle\rangle, \bot \nvdash P' : \{\mathsf{signal} \Leftarrow \mathsf{True}\}}$$

Comparing the derivations of P' and P, you will notice that the parallel expression in P' does not always publish: it publishes only if $l[\bot] \vee \neg h[\top]$. Since this condition depends on high-security information, subsequent expressions must execute in a high-security context. Since `writeL`'s type indicates that it should not be called in a high-security context, the program is insecure.

5 Related Work

The area of information flow security has been thoroughly studied; a survey of recent work is given by Sabelfeld and Myers [11].

Type systems Our approach follows Volpano et al [13], who were the first to propose using type systems to analyze secure information flow. Smith [12] later proposed a type system for concurrent programs which has some important similarities to ours. In Smith's type system, program statements have two labels: one gives an upper bound on the security context of the statement, and the other gives an upper bound on the information which affects the statement's running time. The first is equivalent to the security context in our callable types, and the second to the label of publication conditions in our publication sets. The key difference is that our type system tracks not just the labels of publication conditions, but also their abstract values. This additional complexity is necessary because Orc lacks control-flow statements with explicit scope, so we use publication conditions to detect when the security context can be safely downgraded.

Zdancewic and Myers [15] describe a secure type system for a concurrent language based on the join calculus. Like Orc, the join calculus does not have explicit control flow statements but instead models control flow using communication primitives — in the case of join calculus, channels. This obscures critical facts about program flow. For example, consider a low-security statement which is executed as soon as either of two high-security statements complete. If we know that exactly one of the high-security statements will complete (for example, they correspond to the two possible values of a boolean variable), then this program is secure, since the low-security statement always executes regardless of high-security information. To represent knowledge about statements which always execute exactly once, Zdancewic and Myers introduce the concept of *linear channels* which can be statically guaranteed to receive exactly one value. Program authors must use linear channels explicitly in order to facilitate static analysis. In contrast, our type system for Orc attempts to automatically infer expressions which are always executed based on an abstract interpretation of the program.

Compositionality One important consideration in any static security analysis is compositionality: if two programs are verified independently, will they continue to be secure if they are combined into a larger program? Bossi et al [1] argue that compositionality is an important property of concurrent programs, give compositional criteria for concurrent information flow security, and prove their criteria correct using bisimulation. One advantage of a type system is that, since it constructs the type of an expression recursively from the types of its subexpressions, it is naturally compositional. For example, our type system trivially guarantees that verified programs are always secure under parallel composition.

Dynamic security labels Zheng and Myers [17] describe a secure type system for sequential programs with dynamic security labels. Dynamic security labels enable the set of security principles to be open, so that new security principles (and associated security labels) can be added at runtime. This is very important for workflows, where it is useful to express policies such as "every supervisor can view their employee's records" without having to reverify the workflow every time a new employee joins the company. While Zheng and Myers' approach is formulated in the context of a sequential programming language, it appears to be essentially orthogonal to the underlying programming model. In other words, it should work equally well with a concurrent language such as Orc. Therefore integrating dynamic security labels along the lines suggested by Zheng and Myers with our type system is an important piece of future work.

6 Conclusion

We have sketched a formal method for verifying the security of information flow in workflows. Our approach involves encoding the workflow in the Orc calculus and using a secure type system to reason about the security of the encoded program. The unique features of Orc result in a somewhat unconventional type system which incorporates dependent typing and abstract interpretation in order to reason about termination properties. The resulting combination of language and type system is not significantly more expressive than previous work, but maintains the advantages of Orc in terms of cleanly expressing structured concurrency. This serves as a proof of concept that it is possible to reason about secure information flow properties in Orc programs.

Future work includes proofs of subsumption and non-interference for Orc programs which are well-typed under the proposed secure type system. Once the type system has been proven correct, we intend to extend it with features necessary for practical applications, such as dynamic security labels and limited type inference. The type checker will be implemented as an extension to the Orc language [8] and integrated with the current Orc compiler and runtime environment, enabling users to verify and run workflows expressed in Orc.

Acknowledgements

I would like to thank the members of the Orc research group — David Kitchin, William Cook, Jayadev Misra, Andrew Matsuoka and John Thywissen — for helpful discussions about secure information flow in general and its application to Orc in particular.

References

[1] A. Bossi, C. Piazza, and S. Rossi. Compositional information flow security for concurrent programs. *J. Comput. Secur.*, 15(3):373–416, 2007.

[2] W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in Orc. In *Proc. of the International Conference on Coordination Models and Languages (COORDINATION)*, 2006.

[3] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[4] A. A. et al. Business Process Modeling Language (BPML) 1.0, 2002.

[5] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[6] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), 1998.

[7] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR*, pages 477–491, 2006.

[8] D. Kitchin, A. Quark, W. Cook, and J. Misra. The Orc programming language. In *Proceedings of FMOODS/FORTE*, 2009.

[9] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, May 1999.

[10] J. Misra. Computation orchestration: A basis for wide-area computing. In M. Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, NATO ASI Series, Marktoberdorf, Germany, 2004.

[11] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, (21):2003, 2003.

[12] G. Smith. A new type system for secure information flow. *Computer Security Foundations Workshop, IEEE*, 0:0115+, 2001.

[13] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[14] S. Zdancewic. Challenges for information-flow security. In *1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.

[15] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *CSFW*, pages 29–, 2003.

[16] N. Zeldovich and S. Boyd-Wickizer. Making information flow explicit in histar. *OSDI '06*, pages 263–278, 2006.

[17] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Secur.*, 6(2):67–84, 2007.