Limiting Information Leakage in Event-based Communication

Willard Rafnsson
Chalmers

Andrei Sabelfeld
Chalmers

Abstract

Event-based communication is a major source of power and flexibility for today's applications. For example, in the context of a web browser, the dynamism of user experience is driven by events: fine-grained interaction of the user with a web application triggers events reactively handled by JavaScript code. This paper explores channels for leaking sensitive information through constructs in a reactive language. We propose a general and realizable security framework for preventing information leaks in a reactive setting with such features as new handler creation and hierarchical event structures. While prior work largely takes an all-or-nothing approach to information flows due to intermediate output, our framework tightly regulates the bandwidth of such flows: at most $\log(n+1)$ bits are allowed to be released, where n is the number of public inputs to the program. We gain flexibility from distinguishing between the security levels of message existence and content. A combination of flow-sensitive analysis and buffering output enables us to enforce security without being overly restrictive.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Security and Protection*]: Information flow controls

General Terms Languages, Security

Keywords Information flow, event model, reactive programming

1. Introduction

Event-based communication is a major source of power and flexibility for today's applications. For example, in the context of a web browser, the dynamism of user experience is driven by events: fine-grained interaction of the user with a web application triggers events reactively handled by JavaScript code. Unfortunately, the power of event-based communication opens up channels for leaking sensitive information. This is a concern where programs operate on data of different levels of sensitivity. For example, a *web mashup* is a web application that integrates several services into a new combined service. Typically, a web mashup contains JavaScript code from different Internet domains integrated into a single page. It is essential that sensitive information such as user clicks or input form data (say, in an online shopping part of the mashup) is not propagated to a third party (say, an advertisement part of the mashup). At the same time, separation and isolation based on safe language subsets and reference monitoring [12, 16, 28, 29, 35] is often too restrictive: isolating Google Maps in a mashup from the rest of the web application renders the map-service mashup useless. Hence, a fine-grained approach is desirable, where *information flow* between inputs and outputs is tracked as it is propagated by program constructs [30, 47]. However, information flow in such a scenario is a delicate problem. In the presence of events, there are channels for leaking information that do not arise in standard programming languages [47]. We illustrate the delicacies with web-based examples, but note that the nature of this problem is general.

PLAS'11 5 June 2011, San Jose, California, USA
Copyright © 2011 ACM [copyright data to be supplied]. . . $10.00

Attacker model We are interested in securing reactive programs that do not possess any secrets initially. However, a program interact with its environment by input and output events. Input events might carry secret information (e.g., reading the content of a cookie in JavaScript). Programs may generate output events that might carry public information (e.g., loading an image from a third-party server). Assuming the attacker observes (or controls) public input, the attacker's goal is to deduce information about secret inputs from public outputs. In this model, the only attacker-observable behavior is public output. Internal program behavior such as variable assignment and (non)termination are invisible from outside.

Tracking information flow Some events are more secret than others, e.g., user clicks in an online banking application might need to be protected, while clicks in an online shopping application can be released to a statistics service. The challenge is not to release too much: the fact that a user has submitted a credit-card form can be released, but the credit card number must stay secret. We thus distinguish between the security level of event *existence* and *content*. In the former example, both are secret, but in the latter the existence is public and content is secret. Our model is similar to security labels for structured datatypes [36, 37].

In a standard reactive language, an event triggers a single handler. In a more general setting, a single event might lead to triggering several handlers in an event hierarchy. Coming back to the web setting, an event hierarchy is induced by the *Document Object Model (DOM)* [22] tree, a language-independent interface that regulates access to the tree structure of the underlying HTML document. For example, it is possible to set `onclick` handlers in both the `body` part and a `div` part inside the body. In the event of a click inside the `div` part, both will be triggered and run in sequence.

Balancing security and permissiveness Motivated by the scenario of running potentially malicious JavaScript in a browser, we assume the code is in the hands of the attacker. Hence, all possible channels of information leaks by malicious code need to be addressed. The baseline security condition of *noninterference* [10, 19] prescribes independence of public output from secret input. In a reactive setting, the possibility of observing intermediate outputs needs special attention as it allows high-bandwidth leakage of secrets to the attacker. To this end, existing baseline security conditions in a setting with communication primitives offer two choices of treating intermediate output. We use the terminology of *progress-(in)sensitivity* to highlight the difference. Progress-sensitive noninterference (PSNI) (e.g., [3, 38]) demands that the sequence of outputs produced by programs is fully independent of secrets. This is a strong guarantee, which comes at a price of restrictiveness when enforcing it: Typically, looping on secret data is disallowed [53]. At the other extreme is progress-insensitive noninterference (PINI) (e.g., [2, 3, 7]) that allows programs looping on secret data as long as there are no public side effects in the body loop. However, PINI is vulnerable to brute-force attacks. Consider the source for the following simple web page in Figure 1, where function `brute` is based on an example by Askarov et al. [2].

```
<html> <head> <script type="text/javascript">
function out(v) {
  req=new XMLHttpRequest();
  req.open("GET","a.php?guess=" + v,false);
  req.send(); }
function save() {
  h = document.getElementById('secret').value; }
function brute() {
  l = 0 ;
  while(1){
    out(l) ;
    while(l==h){} ;
    l = l + 1 ;
  } ; }
</script> </head> <body> <center>
<input type="text" id="secret"/>
<input type="button" value="Save"  onclick="save()"/><br/>
<input type="button" value="Brute" onclick="brute()"/>
</center> </body> </html>
```

Figure 1: Brute-force attack in JavaScript

Assume h, `secret` and `save`-clicks are secret and `brute`-clicks public. This web page lets the user `save` a secret value in variable h, and then have the program `brute`-force the value stored on h by successively guessing from 0 to h. Note that there is no explicit passing of sensitive data to the adversary in the code. Nevertheless, when this script diverges, it has already sent the value from h to a server-side script `a.php` (through GET-attribute `guess`) which can then log it for the world to see. This problematic program is deemed secure by PINI and enforcement mechanisms for it [2, 3, 7]). The overrestrictiveness of PSNI and entire-secret leaks of PINI currently leave no choice for anything in-between.

This motivates the need for deeper understanding of security specification and enforcement for reactive languages. While the main long-term motivation for our work is the reactive part of JavaScript in a browser, our results are general and applicable to languages with various flavors of intermediate output. Our results are particularly relevant to languages that feature events, like Erlang, Java, and Smalltalk. Once we gain fundamental understanding of the impact of events, we are in a good position to advance implementation and practical evaluation in a browser setting.

The paper presents the following contributions to securing information flow in event-based systems:

Security framework We introduce a general framework for reasoning about security of reactive programs. A novel contribution is a security framework that addresses the challenge of adequately treating intermediate output. Our security condition occupies the sweet spot between the restrictive PSNI and leaky PINI. It is more restrictive than the latter (preventing brute-force leaks) and more permissive than the former (allowing loops on high data). The condition is a form of *noninterference* [10, 19], that builds insensitivity to computation progress into phases of computation between public inputs. The condition requires that once a public input is consumed, no matter what the secret inputs to the system are, there are only two outcomes until the system is ready to consume another public input: either silent divergence or convergence with the same public output. Thus, *a reactive system that diverges while handling an observable phase handles that phase silently*. Our approach enables tight control over the bandwidth of allowed leaks by connecting it to the number of processed inputs: we show at most $\log(n+1)$ bits are allowed to be released, where n is the number of public inputs to the program. Thus, by controlling the number of public inputs to be processed, we have full control of the amount of released information. This is a major improvement over PINI, where there is no bound on how much information is leaked when handling a single input. Further, the framework includes the possibility for each channel to distinguish between the security levels of message presence and message content. We then develop a JavaScript-like language with such features as new handler creation and hierarchical event handling, to model and analyze code-in-a-browser in this framework. We model a general notion of a hierarchy that includes such tree-like structures as the DOM tree in browsers.

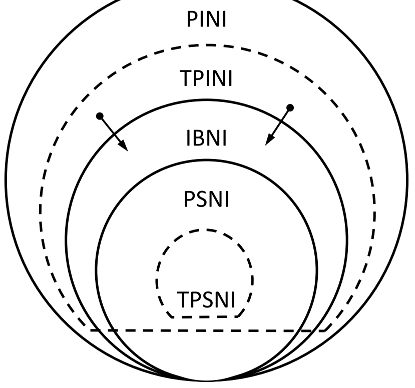


Figure 2: Relative permissiveness of enforcement

Permissive enforcement We support the language with permissive enforcement based on a novel combination of static analysis and transformation. One source of permissiveness is *flow sensitivity*. Our static analysis computes *a mapping from each sink (output channel) to the set of sources (input channels) from which input can leak on that sink*. Another important source of permissiveness is *output buffering*, realized as a transformation that replaces outputs by appending to a queue and flushes the queue immediately before getting ready to receive new input. This transformation removes information leaks from intermediate observations as in the example in Figure 1. Further, we show that all potentially leaking programs that satisfy PINI are *repaired* by buffering. Buffering output to protect against brute-force attacks is the main thrust of our work, and we expect it have most practical consequences.

The set-inclusion diagram in Figure 2 illustrates the relative permissiveness of our enforcement. Bold circles correspond to the sets of programs that satisfy the increasingly liberal security conditions PSNI, IBNI, and PINI (where IBNI is our *input-bounded* noninterference condition). The dashed shapes correspond to the sets of programs that are certified by type-based enforcement TPSNI and TPINI for PSNI and PINI, respectively. The arrows correspond to buffering: programs that are typable with the type system for PINI are moved by buffering from the set TPINI into the set of IBNI programs. This illustrates that we are able to deal with programs in TPINI (in contrast to the restrictive TPSNI) and at the same time guarantee the security property IBNI (in contrast to the leaky PINI).

The rest of the paper is organized as follows. Section 2 presents a stream-based model for reactive systems. Section 3 motivates and introduces IBNI and gives its quantitative implication. Section 4 presents a simple reactive language with new handler creation. Section 5 presents a sound enforcement mechanism for IBNI in this language. Section 6 discusses related work. Section 7 contains conclusions and directions for future work.

2. Stream model

Our goal is to secure information flow in systems producing intermediate output. We address this issue in an incarnation of a gradually-maturing stream-based security model for reactive systems [2, 7, 9, 38]. Here, information can only enter and exit our systems through channel-based message passing. Each channel comes with a label expressing the *confidentiality* level of the information it carries. We then compare each possible input sequence to the resulting output sequence and ensure that confidential information in inputs does not leak into public outputs. An important issue this model deals with is that of *feedback loops*. Since some inputs can be generated as a function of outputs, it would seem that we have to consider the behavior of the environment when performing information flow security analysis on our systems, like in [38]. However, as proved in [9], for deterministic programs, it is sufficient to consider only input sequences which are independent of outputs, as quantifying over all these in our security conditions will necessarily include dependent input sequences. This yields compositional results, as we do not have to take into account the behavior of the environment. A sequence of inputs or outputs can then be given as a single *stream*, i.e., a (possibly infinite) list of messages. We assume that the environment supplies an input and output buffer for the input and output stream, thus making the communication between our reactive system and its environment asynchronous. This greatly simplifies our framework, as a reactive system can be considered as a *stream transducer*, transforming a given input stream I into an output stream O, much like a batch-job program transforms an initial memory to a final memory, a scenario thoroughly explored in information-flow security [47]. Still, there is a key difference from batch-job computation: the possibility of producing intermediate outputs. We will return to this difference and show how to secure the information-flow channel (progress) it introduces.

This model appears in its most mature form in [7], and it is this model ours resembles most. Like [7], we treat deterministic reactive systems which operate on streams. However, instead of defining streams and relations on these coinductively, our treatment of streams resembles the one used in Scheme and Haskell, and relations on streams are defined inductively. Furthermore, our security policies are more fine-grained, distinguishing the confidentiality level of message existence and content.

2.1 Reactive systems

Our computation model is that of *reactive systems*, in which computation occurs as a reaction to an external *event*. These events, which could e.g. represent a keystroke, GUI button click, network packet reception, sensor reading, or timer event, are triggered in the system by the *environment* in which the system runs. This environment could for instance consist of users, hardware, or other systems, such as a web browser, as in our setting where the reactive system is a JavaScript program. Indeed, as exemplified by a web browser running in an environment consisting of a user and other computers on the network, a reactive system can itself be a reactive system running in an even greater environment. While reacting to an event, a reactive system can change its state, as well as trigger zero or more events in its environment. This *interaction* of a reactive system with its environment is modeled by channel-based message-passing. Each event the system reacts to is associated an input channel, and the environment triggers a given event in the system by sending a message, containing a value, to the system on the associated channel. Likewise, the system triggers events in its environment by sending messages on output channels. Inputs i, outputs o, and messages s are then given by

$$i ::= ch(v) \qquad o ::= \bar{ch}(v) \mid \bullet \qquad s ::= i \mid o$$

where $ch(v)$ (resp. $\bar{ch}(v)$) denotes a message received (resp. sent) on channel ch carrying value v, and $\bullet$ denotes that a silent, internal step, or "tick" occurred in the source of the $\bullet$ (e.g. an internal channel synchronization, memory assignment, etc.). These channels are the only external interface to the reactive system, and therefore, the only medium by which information can enter and exit our systems.

The behavior of a reactive system can now be given by a *labeled transition system* with actions ranged by i and o. That is, a triple

$$(Q, A, \{\xrightarrow{a} \mid a \in A\}),$$

where Q is a set of *states*, A a set of *actions*, and $\xrightarrow{a} \subseteq Q \times Q$, for all $a \in A$. Intuitively, if q is a state which can, as its next computation step, input i and enter state q', then $q \xrightarrow{i} q'$. Likewise, $q \xrightarrow{o} q'$ if q can output o and enter state q' as its next step. Practical computation models native to this paradigm include event loops, actors in the Actor Model, and, of interest here, JavaScript programs.

2.2 Streams

Consider the classic list operators cons, head and tail, given by

$$\mathsf{cons}(x, X) = x :: X \quad \mathsf{head}(x :: X) = x \quad \mathsf{tail}(x :: X) = X.$$

Here, $(::)$ is a data constructor, where $x :: X$ represents X with x prepended (or "cons"ed). For languages with lists, we define the *next* operator $\triangleright$ to evaluate a term X *until* it reaches the form $x :: X'$ for some terms x and X'[1]. Without further evaluating x and X', $\triangleright$ then yields $x :: X'$. So, $X \triangleright x :: X'$. For example, given

$$\mathsf{inc}(n) = n :: \mathsf{inc}(n+1),$$

$\mathsf{head}(\mathsf{inc}(3))$ will first evaluate to $\mathsf{head}(3 :: \mathsf{inc}(3+1))$ since $\mathsf{inc}(3) \triangleright 3 :: \mathsf{inc}(3+1)$. At this point, with an appropriate evaluation strategy, $\mathsf{head}(3 :: \mathsf{inc}(3+1))$ can evaluate directly to 3, without first evaluating $\mathsf{inc}(3+1)$ to a value. This idea of reducing a term only *as needed* (i.e. lazily) to yield the head-and-tail of a list exists as streams in Scheme and lists in Haskell, with which you can express finite *or infinite* lists. Now, S is a (non-empty) *message stream* if $S \triangleright s :: S'$ for some s and S'. If $\triangleright$ is not defined on S, then S is an *empty* message stream, denoted by the empty list symbol $[]$. Input streams I and output streams O are defined similarly. Throughout the paper, we frequently denote by $s :: S'$ any stream S for which $S \triangleright s :: S'$.

How do we compare possibly infinite lists? We use the idea that two streams are equivalent if they cannot be distinguished. S_1 and S_2 are distinct, written $S_1 \doteqdot S_2$ if a component-wise equality check of S_1 and S_2 eventually[2] fails. $S_1 \doteqdot S_2$ is defined in Figure 3. Throughout the paper, if a rule is labeled with $(*)$ on its right, then we have, for brevity, neglected to write the symmetric counterpart of the $(*)$-labeled rule into the definition (to obtain the symmetric counterpart of the $(*)$-labeled rule in Figure 3, swap $s_1 :: S_1$ and $[]$). We then define stream equivalence as

$$S_1 \equiv S_2 \stackrel{\text{def}}{=} \neg(S_1 \doteqdot S_2).$$

2.3 Runs as streams

Viewed externally, a *run* (trace) of a reactive system state q on an input stream I, denoted $q(I)$, is a sequence of messages consisting

[1] That is, $x :: X'$ is the *head normal form* of X.

[2] After a finite number of equality checks.

$$\frac{s_1 \neq s_2}{s_1 :: S_1 \doteqdot s_2 :: S_2} \qquad \frac{s_1 = s_2 \quad S_1 \doteqdot S_2}{s_1 :: S_1 \doteqdot s_2 :: S_2}$$

$$\frac{-}{s_1 :: S_1 \doteqdot []}(*)$$

Figure 3: Distinction of streams.

of the inputs in I interleaved with the outputs emitted while q reacts to each input. We interpret a run as a message stream in Figure 4 by defining $\triangleright$ on runs. While this definition allows runs to be nondeterministic, we assume $q(I)$, and thus q, to be deterministic, as we are ultimately interested in the reactive part of JavaScript (which is deterministic and single-threaded). However, the step to nondeterminism in our results is small: Nondeterministic choice can be resolved through a labeled reduction. By giving a random choice stream to a run, we effectively "factor out" nondeterminism into streams, as per O'Neil et al. [38].

$$\frac{q \xrightarrow{i} q'}{q(i :: I) \triangleright i :: q'(I)} \qquad \frac{q \xrightarrow{o} q'}{q(I) \triangleright o :: q'(I)}$$

Figure 4: Next operator for a run.

When we are only interested in the outputs in a stream, we re-interpret the stream, using the definition of $\triangleright$ in Figure 5. The re-interpretation can be viewed as an operator $(\cdot_{\mathsf{o}})$ which "filters" (by need) the inputs from a stream[3], yielding the outputs.

$$\frac{S_{\mathsf{o}} \triangleright o :: O}{(i :: S)_{\mathsf{o}} \triangleright o :: O} \qquad \frac{-}{(o :: S)_{\mathsf{o}} \triangleright o :: S_{\mathsf{o}}}$$

Figure 5: Outputs in a stream.

When defining security for reactive systems, we need only consider input streams of finite length. Assume a given infinite input stream causes our system to leak. Then there must be a finite input stream which causes the same leak. Otherwise, the "attack" requires infinite consumption to succeed, in which case the attack never finishes. Similarly, we only need to consider messages which are finitely far into the output stream. Therefore it suffices to define conditions on streams in security definitions inductively.

In summary, a stream is a possibly infinite list of messages. A run $q(I)$ of a reactive system q on an input stream I can be seen as a possible interaction of q with an environment that feeds I to q. In that case, q transduces I to the output stream $O = (q(I))_{\mathsf{o}}$.

3. Security of reactive systems

We now formalize a notion of information security which rejects leaking systems. As mentioned earlier, the observables of a reactive system are its inputs and outputs. Intuitively, if an I is changed in a way that cannot be observed, then there must be no observable difference in the resulting O. This intuition corresponds to the notion of noninterference [9, 10, 19].

Whether a message on a channel is observable or not is indicated by a *security label* associated with the channel. We assume a lattice of security levels $(\mathcal{L}, \sqsubseteq)$ expressing levels of confidentiality. In our examples, $\mathcal{L} = \{H, L\}$ and $\sqsubseteq = \{(H, H), (L, H), (L, L)\}$, with H for "high" and L for "low" confidentiality. We let $\mathsf{lbl}(ch)$, the security label we associate with a channel ch, be a pair of security levels from $\mathcal{L}$. Here, if $\mathsf{lbl}(ch) = l_{\mathsf{c}}^{l_{\mathsf{e}}}$, then l_{c} is the confidentiality of values (content) passed on ch, and l_{e} the confidentiality of the existence of a message on ch. For instance, a channel carrying secret values but where the presence of messages is public has label H^L. We note that $l_{\mathsf{e}} \sqsubseteq l_{\mathsf{c}}$, since being capable of observing values on ch necessarily implies being capable of observing that *some* message was transmitted on ch. So L^H is impossible. We abbreviate channel labels H^H, H^L and L^L by H, M and L, respectively. In our examples we denote a channel by its label when its name does not matter. We let $\mathsf{lbl}(ch(v)) = \mathsf{lbl}(\bar{ch}(v)) = \mathsf{lbl}(ch)$. When $\mathsf{lbl}(\bullet) = l_{\mathsf{c}}^{l_{\mathsf{e}}}$, then $l_{\mathsf{c}} = l_{\mathsf{e}}$, which in our examples equals H. The distinction of existence and content levels is similar to that for general datatypes. For example, Jif [36, 37] allows arrays, where the length of the array is public but the individual elements are secret.

The security labels express *who* can observe *what*. An observer is associated a security label l from $\mathcal{L}$, indicating the observer is capable of observing the value in a message s with $l_{\mathsf{c}} \sqsubseteq l$, and the presence of a message with $l_{\mathsf{e}} \sqsubseteq l$, where $\mathsf{lbl}(s) = l_{\mathsf{c}}^{l_{\mathsf{e}}}$. The l-observables in s, $\mathsf{obs}_l(s)$, are thus

$$\mathsf{obs}_l(\bullet) = \begin{cases} \bullet & \text{if } l_{\mathsf{e}} \sqsubseteq l \\ \cdot & \text{otherwise} \end{cases}$$

$$\mathsf{obs}_l(ch(v)) = \begin{cases} ch(v) & \text{if } l_{\mathsf{c}} \sqsubseteq l \\ ch(\cdot) & \text{if } l_{\mathsf{c}} \not\sqsubseteq l \wedge l_{\mathsf{e}} \sqsubseteq l \\ \cdot & \text{otherwise} \end{cases}$$

$\mathsf{obs}_l(\bar{ch}(v))$ defined in the same manner as $\mathsf{obs}_l(ch(v))$

where $\cdot$ means nothing is observed[4]. We also use $\mathsf{obs}_l(s)$ as a predicate, where $\mathsf{obs}_l(s)$ is false when $\mathsf{obs}_l(s) = \cdot$ and true otherwise. Also, we define $s_1 =_l s_2$ iff $\mathsf{obs}_l(s_1) = \mathsf{obs}_l(s_2)$.

We are most interested in observable messages in message streams in our security definitions. This motivates the stream re-interpretation in Figure 6, which can be viewed as an operator $(\cdot_l)$ which "drops" unobservables from a stream, until an observable is found[5]. We denote by $s ::_l S'$ any stream S for which $S_l \triangleright o :: S'$.

$$\frac{\neg\mathsf{obs}_l(s) \quad S_l \triangleright s' :: S'}{(s :: S)_l \triangleright s' :: S'} \qquad \frac{\mathsf{obs}_l(s)}{(s :: S)_l \triangleright s :: S}$$

Figure 6: Next l-observable in a stream

We say S is l-silent, written $\mathsf{sil}_l(S)$, when S produces no l-observables, that is, when $S_l \equiv []$. The finite stream predicate, $\mathsf{fin}(S)$, is defined Figure 7.

$$\frac{-}{\mathsf{fin}([])} \qquad \frac{\mathsf{fin}(S)}{\mathsf{fin}(s :: S)}$$

Figure 7: Finite stream predicate

If we were not interested in unobservables at all, we would use the re-interpretation $(\cdot_l^!)$ in Figure 8 which filters *all* unobservables from a stream. However, $(\cdot_l^!)$ hides whether a stream is silent and finite or silent and infinite. It is this subtle detail which is of key importance in the distinction between the security definitions we study in the next section.

[3] Like "filter" in Python, Erlang and Haskell.

[4] The observer only learns that a message occurred on ch by observing $ch(\cdot)$ or $\bar{ch}(\cdot)$. The observer knows the specification of the reactive system, so this might enable the observer to infer on the reactive system state.

[5] Like "dropwhile" in Python, Erlang and Haskell.

$$\frac{\neg\mathsf{obs}_l(s) \qquad S_l^! \triangleright s' :: S'}{(s :: S)_l^! \triangleright s' :: S'} \qquad \frac{\mathsf{obs}_l(s)}{(s :: S)_l^! \triangleright s :: S_l^!}$$

Figure 8: l-observables in a stream

3.1 Progress and Termination

The choice on how to deal with diverging runs which produce no observables leads to different security conditions. They differ in whether or not they secure *progress* and *termination* observations. An observer capable of observing termination will know, by inspecting its observables, whether a program is currently diverging or not. Such an observer would be capable of learning whether h is "true" or not in the following program.

$$\texttt{in } M(h)\,;\ \texttt{while } h\,\{\texttt{skip}\} \tag{1}$$

This program inputs a (secret) value on M, binds it to h, and then loops on h. Noninterference definitions which secure such observations are *termination-sensitive*, and those that do not are *termination-insensitive*. As we mention in the introduction, the attacker in our reactive setting does not observe divergence because of its internal nature; observers can only reason about the behavior of our systems by observing message transmissions. An observer capable of observing progress, on the other hand, will know how far a program is in its computation. Such an observer would learn the value of h in the following program.

```
in $M(h)$; $l$ := 0;
while $l <= h$ {
  out $L(l)$; $l$ := $l + 1$ }                    (2)
```

After assigning an input v on M to h, this program outputs the sequence of numbers $0..v$ on L. Observers *not* capable of observing progress would, upon observing an output sequence $[L(0), L(1), L(2)]$ not know whether $L(2)$ is the final output of the program (meaning $h = 2$), or whether $L(3)$ will follow. Thus, the observer would never know the exact value of h. Noninterference definitions which secure these observations are *progress-sensitive*, and those that do not are *progress-insensitive*.

Bohannon et al. [7] define several noninterference notions, and note two of them to be of practical interest. As they coincide on finite streams, the interesting bit is how they treat infinite streams, in particular, streams which eventually become silent and infinite. The first definition, ID (indistinguishable) security, is given as

Definition 3.1. *q is* ID-secure *iff, for all l, I_1 and I_2,*

$$I_1 \sim_l I_2 \implies (q(I_1))_\circ \sim_l (q(I_2))_\circ,$$

where $S_1 \sim_l S_2 \overset{\text{def}}{=} \neg(S_1 \dot{\sim}_l S_2)$.

$$\frac{s_1 \neq_l s_2}{s_1 ::_l S_1 \dot{\sim}_l s_2 ::_l S_2} \qquad \frac{s_1 =_l s_2 \qquad S_1' \dot{\sim}_l S_2'}{s_1 ::_l S_1 \dot{\sim}_l s_2 ::_l S_2}$$
$$\frac{\mathsf{sil}_l(S_2) \qquad \mathsf{fin}(S_2)}{s ::_l S_1 \dot{\sim}_l S_2}(*)$$

Figure 9: ID-difference of streams

$(\ \dot{\sim}_l\ )$ is given in Figure 9. Intuitively, for $S_1 \sim_l S_2$ to hold, the l-observables of S_1 and S_2 must be component-wise equal, until either a) both S_j have no more l-observables, or b) one S_j is silent and infinite. ID-security rejects programs like (2). One might therefore be lead to believe that ID-security is progress-sensitive. However, by exploiting the exception in b), (2) can leak all of h when progress is observable, as follows.

```
in $M(h)$; $l$ := 0;
while $l <= h$ {
  out $L(l)$; $l$ := $l + 1$                      (3)
}; while 1 {skip}
```

In fact, program (3) has the same input-output behaviour as the brute-force attack in Figure 1, extracted in program (4).

```
in $M(h)$; $l$ := 0;
while 1 {
  out $L(l)$;                                     (4)
  while $l = h$ {skip};
  $l$ := $l + 1$ }
```

ID-security is thus both progress- and termination-*insensitive*. One might disregard this issue, thinking that, while there are leaky PINI-secure programs, PINI-enforcements will surely reject them. However, the way programs like (4) exploit the "progress channel" cannot be detected by current PINI-enforcements, since they contain no explicit leaks of h or L effects in a H context (loop/branch).

The other definition noted to be of practical interest in [7] is CP (co-productive) security, defined as follows.

Definition 3.2. *q is* CP-secure *iff, for all l, I_1 and I_2,*

$$I_1 \simeq_l I_2 \implies (q(I_1))_\circ \simeq_l (q(I_2))_\circ,$$

where $S_1 \simeq_l S_2 \overset{\text{def}}{=} \neg(S_1 \dot{\simeq}_l S_2)$.

$$\frac{s_1 \neq_l s_2}{s_1 ::_l S_1 \dot{\simeq}_l s_2 ::_l S_2} \qquad \frac{s_1 =_l s_2 \qquad S_1' \dot{\simeq}_l S_2'}{s_1 ::_l S_1 \dot{\simeq}_l s_2 ::_l S_2}$$
$$\frac{\mathsf{sil}_l(S_2)}{s ::_l S_1 \dot{\simeq}_l S_2}(*)$$

Figure 10: CP-difference of streams

$(\ \simeq_l\ )$ is given in Figure 10. Observe the minute, yet key, difference between $(\ \dot{\simeq}_l\ )$ and $(\ \dot{\sim}_l\ )$. Intuitively, for $S_1 \simeq_l S_2$ to hold, the observables of S_i must match exactly. Indeed, we have $S_1 \simeq_l S_2 \iff S_{1l}^! \equiv S_{2l}^!$. CP-security rejects programs like (4), and is thus progress-*sensitive*. It will, however, accept programs like (1), where the only (possibly) observable behavioral difference is whether the program diverges or not. So CP-security is termination-insensitive.

We have now seen most of Figure 2. ID-security is PINI, the set of programs proven ID-secure through enforcement make up TPINI, CP-security is PSNI and the set of programs proven CP-secure through enforcement make up TPSNI. While ID-security can leak everything, CP-security leaks nothing in our setting since we do not consider the termination channel exploitable[6]. Since leaking arbitrarily on the progress channel is unacceptable in practice, CP-security is a much more reasonable property to aim for. However, CP-security is hard to enforce permissively [53]; typically, looping on high data is disallowed.

3.2 IB-security

What makes the brute force attack successful is that before the program reaches a point in its control flow where it will diverge, the program has already leaked its secret through intermediate outputs. We devise a new security notion, IB-security (input-bounded), which deals with this problem by requiring that *a reactive system that diverges while handling an observable phase handles that*

[6] If the termination status of a program is observable, then CP-security will leak at most 1 bit (per execution) [2] in any case.

phase silently. Phases arise from the idea that an observer might consider it possible for unobservables to appear before any observable he sees in a stream, and after the last observable he sees. If S is silent, all of S is one phase. Otherwise, the first phase of S are all the messages in S up to (and including) the first observable. The next phase is then the first phase in the rest of the stream. Figure 11 partitions a stream this way by placing a $*$ into the stream between phases. Let $\mathsf{o} ::= * \mid o$, $\mathsf{s} ::= i \mid \mathsf{o}$ and S be streams of s. We set $\mathsf{lbl}(*) = \bot^{\bot}$ ($= L$ in our examples), so $\mathsf{obs}_l(*) = *$, for all l.

$$\frac{\neg\mathsf{obs}_l(i)}{(i :: S)_l^{\mathsf{p}0} \triangleright i :: S_l^{\mathsf{p}0}} \qquad \frac{\mathsf{obs}_l(i)}{(i :: S)_l^{\mathsf{p}0} \triangleright i :: S_l^{\mathsf{p}1}}$$

$$\frac{-}{(o :: S)_l^{\mathsf{p}k} \triangleright o :: S_l^{\mathsf{p}k}} \qquad \frac{-}{(i :: S)_l^{\mathsf{p}1} \triangleright * :: (i :: S)_l^{\mathsf{p}0}}$$

Figure 11: Partition stream into observable phases. $S_l^{\mathsf{p}} \overset{\text{def}}{=} S_l^{\mathsf{p}0}$

Definition 3.3. *q is* IB-secure *iff, for all l, I_1 and I_2,*

$$I_1 \ \approx_l \ I_2 \implies (q(I_1))_{\mathsf{o}}^{\mathsf{p},l} \ \approx_l \ (q(I_2))_{\mathsf{o}}^{\mathsf{p},l},$$

where $S_{\mathsf{o}}^{\mathsf{p},l} \overset{\text{def}}{=} (S_l^{\mathsf{p}})_{\mathsf{o}}$ *and* $S_1 \ \approx_l \ S_2 \overset{\text{def}}{=} \neg(S_1 \dot{\approx}_l S_2)$.

$$\frac{\mathsf{s}_1 \neq_l \mathsf{s}_2}{\mathsf{s}_1 ::_l \mathsf{S}_1 \dot{\approx}_l^k \mathsf{s}_2 ::_l \mathsf{S}_2}$$

$$\frac{\mathsf{S}_1 \dot{\approx}_l^0 \mathsf{S}_2}{* ::_l \mathsf{S}_1 \dot{\approx}_l^k * ::_l \mathsf{S}_2} \qquad \frac{s_1 =_l s_2 \quad \mathsf{S}_1 \dot{\approx}_l^1 \mathsf{S}_2}{s_1 ::_l \mathsf{S}_1 \dot{\approx}_l^k s_2 ::_l \mathsf{S}_2}$$

$$\frac{\mathsf{sil}_l(S_2) \quad \mathsf{fin}(S_2)}{\mathsf{s} ::_l \mathsf{S}_1 \dot{\approx}_l^0 \mathsf{S}_2}(*) \qquad \frac{\mathsf{sil}_l(S_2)}{\mathsf{s} ::_l \mathsf{S}_1 \dot{\approx}_l^1 \mathsf{S}_2}(*)$$

Figure 12: IB-difference of partitioned streams. $\dot{\approx}_l \overset{\text{def}}{=} \dot{\approx}_l^0$

($\dot{\approx}_l$) is given in Figure 12. IB-difference behaves like ID-difference (this is ($\dot{\approx}_l^0$)) until an observable message is found in both S_1 and S_2; then it behaves like CP-difference (this is ($\dot{\approx}_l^1$)). As soon as a $*$ is observed in both S_1 and S_2, however, IB-difference goes back to behaving like ID-difference. In both these cases, observable messages, and $*$, have to match.

IB-security rejects Program (3). For instance, let $I_1 = [H(1)]$ and $I_2 = [H(2)]$, each an input stream with a single phase. Running (3) on these streams yields outputs that are IB-equivalent to the lists $O_1 = [L(1), \uparrow]$ and $O_2 = [L(1), L(2), \uparrow]$, respectively (where $\uparrow$ denotes silent divergence). However, $O_1 \dot{\approx}_l O_2$ since, after matching $L(1)$, one stream is silent and the other is not.

It should not be too surprising that IB-security resides between ID-security and CP-security. The proofs of these and further formal results are given in the full version of this paper [40].

Proposition 3.1. *If q is CP-secure, then q is IB-secure.*

Proposition 3.2. *If q is IB-secure, then q is ID-secure.*

IB-security does not stop progress leaks entirely. For instance, the "guess" attack in (5) is IB-secure while an observer can learn the correctness of his guess by probing the responsiveness of (5).

```
in $H(h)$;
while 1 {
  in $L(l)$;                                   (5)
  while $l = h$ {skip};
  out $L(l)$; }
```

A JavaScript modeling of this program is given in Figure 13. Here,

```
<html> <head> <script type="text/javascript">
/* functions out(v) and save() same as in Figure 1 */
function guess(v) {
  while(v==h){};
  out(v);
}
</script> </head> <body> <center>
<input type="text" id="secret"/>
<input type="button" value="Save" onclick="save()"/> <br>
<input type="text" id="public"/>
<input type="button" value="Guess"
onclick="guess(document.getElementById('public').value)"/>
</center> </body> </html>
```

Figure 13: Guess attack in JavaScript

the guess is fed by the user through (public) clicks on a `guess` button[7]. The key difference between programs (3) and (5) is that program (5) leaks only "a little" as a reaction to each phase, and thus has a lower *bandwidth* on the progress channel. A crucial question arises: What is the maximum bandwidth of leaks IB-security permits on the progress channel? We answer this question in the following section.

3.3 Quantitative guarantee

Our security condition entails a tight quantitative security guarantee. We utilize Smith's recent model for quantitative security. Smith [51] defines the notion of *vulnerability* $V(X)$ as the worst-case probability of guessing the value of secret X by an adversary in one try. The measure of information quantity is then defined as $-\log V(X)$, which corresponds to *min-entropy*. Based on the intuition

information leaked = initial uncertainty − remaining uncertainty,

Smith defines *information leakage*, which for deterministic programs and uniformly distributed secrets amounts to $\log |S|$, where $|S|$ is the size of the set of possible public outputs S given that the public input is fixed. $|S|$ translates to the number of indistinguishability classes for the high input, which, in effect, is the number of different possibilities for the phases of an input stream. This is also in line with Lowe [27], who measures the number of secret behaviors distinguished by an attacker in a nondeterministic setting.

Smith's model allows us to obtain a quantitative guarantee without reasoning about probabilities. Indeed, it suffices to give an estimate on the number of possible public observations in order to derive min-entropy. For the quantitative results, we assume input streams are drawn from a finite universe $\mathcal{U}(I^L)$, where I^L is a (fixed) stream of observables where $I^L \equiv I_l^!$ holds for each $I \in \mathcal{U}(I^L)$. Given that the number of input streams satisfying this criteria is infinite, and that we thereby seemingly lose precision by assuming a finite universe, we note that the result which is based on this assumption holds *regardless of how we fix our finite universe*.

Let E be an equivalence relation, $[a]_A^E$ the E-equivalence class of a in A, and A/E the set of E-equivalence classes in A. Formally,

$$[a]_A^E \overset{\text{def}}{=} \{b \in A \mid (a, b) \in E\}$$
$$A/E \overset{\text{def}}{=} \{[a]_A^E \mid a \in A\}.$$

Definition 3.4 (k-bit security)**.** *Let I^L and $\mathcal{U}(I^L)$ be given, $\mathcal{U}(I^L)$ uniformly distributed, and q be a system taking input from $\mathcal{U}(I^L)$. Then q is k-bit secure if $k \leq \log_2 |S|$, where*

$$q(\mathcal{U}(I^L)) \overset{\text{def}}{=} \{(q(I))_{\mathsf{o}} \mid I \in \mathcal{U}(I^L)\}$$
$$S \overset{\text{def}}{=} q(\mathcal{U}(I^L)) / \simeq_l$$

[7] A similar example can be made where the guess comes from the network.

In our setting, a k-bit secure program leaks at most k bits. The following program leaks whether the first value received on the M-channel (if any) is even or odd.

$$\texttt{in } M(h)\texttt{; while } (h\texttt{\%}2)\ \{\texttt{skip}\}\texttt{; out } L(\texttt{0}) \qquad (6)$$

For fixed low inputs in an input stream with at least 1 M-message, the observer sees at most two kinds of outputs: those equivalent to $[]$ and $[L(0)]$ (by $\simeq_l$) respectively. As $\log_2 |S| = \log_2 2 = 1$, Program (6) is at most 1-bit secure. Program (3), on the other hand, eventually outputs the exact value received last on the H-channel. In this case we have at most m possible outputs, where m is the number of integers in $\mathcal{U}(I^L)$. Since $\log_2 |S| = \log_2 m$, and since all these bits come from a single secret input value, Program (3) leaks that whole value. At last, the number of bits leaked by Program (5) is a function of the length of the observables I^L. If I^L has n messages, then I^L has $n + 1$ phases. Depending on the secret, the program can diverge when handling any of these phases, or in none of them. The last phase must be handled silently. We thus have $n + 1$ classes of outputs, so Program (5) is at most $\log_2(n + 1)$-bit secure.

Theorem 3.1. *If q is IB-secure, then q is at most $\log_2(n + 1)$-bit secure, where n is the nr. of observables in q's input.*

3.4 Buffering improves security

The reader might wonder which reactive systems in *general* are IB-secure. It turns out that ID-secure systems which *buffer* outputs between handling of inputs are IB-secure[8]. We give a buffered re-interpretation of a stream in Figure 14, which buffers outputs between each input. Basically, if $S \triangleright o :: S'$ for some o and S', then $S_\mathsf{B} \triangleright o :: S''$ for some S'' *only if* an input follows o in S, or S *is* a finite number of outputs. We realize this idea with a two-mode re-interpretation: buffer (annotation B), and flush (annotation F). $(S, O)_\mathsf{B}$ will, when the next operator is applied on it, queue non-$\bullet$ outputs from S in O using the reverse "cons" constructor[9]. This constructor interacts with the next operator as follows.

$$[] :: s \triangleright s :: [] \qquad (s :: S) :: s' \triangleright s :: (S :: s')$$

When an input is encountered, O is flushed. So, O practically takes over for S until exhausted.

$$\frac{-}{([], o :: O)_\mathsf{B} \triangleright o :: O} \qquad \frac{-}{(\bullet :: S, O)_\mathsf{B} \triangleright \bullet :: (S, O)_\mathsf{B}}$$

$$\frac{o \neq \bullet}{(o :: S, O)_\mathsf{B} \triangleright \bullet :: (S, O :: o)_\mathsf{B}} \qquad \frac{(i :: S, O)_\mathsf{F} \triangleright s :: S'}{(i :: S, O)_\mathsf{B} \triangleright s :: S'}$$

$$\frac{-}{(S, o :: O)_\mathsf{F} \triangleright o :: (S, O)_\mathsf{F}} \qquad \frac{-}{(i :: S, [])_\mathsf{F} \triangleright i :: (S, [])_\mathsf{B}}$$

Figure 14: Buffered stream. $S_\mathsf{B} \overset{\text{def}}{=} (S, [])_\mathsf{B}$

Let q_B be like q in every way, except when run on an input stream I. The resulting stream is then $(q(I))_\mathsf{B}$ instead of $q(I)$.

Theorem 3.2. *If q is ID-secure, then q_B is IB-secure.*

This theorem is central. It states that we can drastically reduce the leak on the progress channel by running the program in a context which buffers output. In practice, however, having the context do this buffering is not always an option; in JavaScript, for instance, this would require changing the JavaScript interpreter. However, in such a scenario, buffering can be realized through program transformation, by "inlining" the buffering into the JavaScript program. Then, provided the JavaScript program can be enforced to be ID-secure, applying the buffering transformation on the program will make it IB-secure. We now give a concrete example of an ID-security enforcement and a buffering program transformation in a JavaScript subset. The language extends the one given in [7], but the enforcement and program transformation are ours.

[8] While it is sufficient to buffer output between handing of observable input phases, doing so is not viable in practise where there might be multiple (unknown) observers at different observation levels (for instance, in a Mashup).

[9] "snoc" in Haskell.

4. Language

We now present a simple core language for reactive imperative systems, given in Figure 15. The language is a subset of JavaScript, sharing many of its features and assumptions. In this language, when reacting to an event, a reactive system runs a *handler* associated with that event, as well as all handlers above it in its hierarchy of event handlers. Each such handler can change the state of the reactive system, and trigger zero or more events in its environment. Abstractly, our systems repeat the following: i) take the next available input, ii) produce zero or more outputs. Inputs are buffered, and then handled in the order they are received in. Our programs are single-threaded in the sense that it does not handle input messages concurrently. Input and output channels are disjoint, so our programs cannot send messages to themselves. This last restriction is not severe; in JavaScript, events generated procedurally are implemented as procedure calls[10]. Besides, we are most interested in how our systems react to their environment.

4.1 Syntax

$$\begin{array}{lll}
p & ::= & \cdot \mid ha; p \\
ha & ::= & ch(\texttt{z})\{c\} \\
c & ::= & \texttt{skip} \\
 & \mid & c;\, c \\
 & \mid & x \texttt{ := } e \\
 & \mid & \texttt{if } e\ \{c\}\ \{c\} \\
 & \mid & \texttt{while } e\ \{c\} \\
 & \mid & \texttt{out } ch(e) \\
 & \mid & \texttt{new } ha
\end{array}$$

Figure 15: Syntax

Let programs, handlers, commands and expressions be ranged by p, ha, c, e, respectively, and let the sets $\mathbb{C}$, $\mathbb{X}$, and $\mathbb{V}$ of channels, variables and values respectively be ranged by ch, x, and v. A program p is a list of handlers. When p processes an input $ch(v)$, it looks through its list of handlers for a ch-handler, $ch(\texttt{z})\{c\}$. If none is found, $ch(v)$ is dropped. If found, p will execute the body of the handler, c, with v in place of the formal parameter $\texttt{z}$. A command is merely a program in a `while` language, extended with *output* and *handler creation*. Beyond memory inspection and modification, branching and looping, c can output messages, and add/replace a handler to/in p. A *memory*, ranged by μ, is a $\mathbb{X} \to \mathbb{V}$ mapping which, initially, is 0 for all x. This memory is global, so when p processes an input, the change in memory can affect how other handlers process future input.

After (if) c terminates, p consults a *hierarchy* of channels H, processing $ch(v)$ as if it were an input to the parent of ch (effectively forwarding $ch(v)$ to the parent of ch). $H(ch)$ yields the parent channel of ch, or $\top$ if ch has no parent. In effect, H is a tree (or a forest) and can be used to model e.g. the DOM tree. Once a message has been forwarded to $\top$, p will enter a state where it is ready to process a new input.

We assume the presence of an expression language, which can be more or less arbitrary, except the relation $\mu \vdash e \Downarrow v$, which under memory μ reduces e to v, must be given. This relation must be side-effect free, deterministic and terminating. $\mathbb{X}$ and $\mathbb{V}$ must be

[10] `timeOut` events are an exception. However, we can model these by considering `setTimeout("`s`", `ms`);` in JavaScript as a request to the browser to send a message on a reserved channel after time ms to the JavaScript, which stands ready with a handler which reacts by running s.

$$\frac{-}{(\mu, p, \texttt{skip};\ c) \xrightarrow{\bullet} (\mu, p, c)}$$

$$\frac{(\mu, p, c_1) \xrightarrow{o} (\mu', p, c_1')}{(\mu, p, c_1\,;\ c_2) \xrightarrow{o} (\mu', p, c_1'\,;\ c_2)}$$

$$\frac{\mu \vdash e \Downarrow v}{(\mu, p, x \texttt{ := } e) \xrightarrow{\bullet} (\mu[x \mapsto v], p, \texttt{skip})}$$

$$\frac{\mu \vdash e \Downarrow v \qquad v \neq 0}{(\mu, p, \texttt{if}\ \ e\,\{c_1\}\,\{c_2\}) \xrightarrow{\bullet} (\mu, p, c_1)}$$

$$\frac{\mu \vdash e \Downarrow 0}{(\mu, p, \texttt{if}\ \ e\,\{c_1\}\,\{c_2\}) \xrightarrow{\bullet} (\mu, p, c_2)}$$

$$\frac{\mu \vdash e \Downarrow v}{(\mu, p, \texttt{out}\ \ ch\texttt{(}e\texttt{)}) \xrightarrow{\bar{ch}(v)} (\mu, p, \texttt{skip})}$$

$$\frac{\mu \vdash e \Downarrow 0}{(\mu, p, \texttt{while}\ \ e\,\{c\}) \xrightarrow{\bullet} (\mu, p, \texttt{skip})}$$

$$\frac{\mu \vdash e \Downarrow v \qquad v \neq 0}{(\mu, p, \texttt{while}\ \ e\,\{c\}) \xrightarrow{\bullet} (\mu, p, c;\ \texttt{while}\ \ e\,\{c\})}$$

$$\frac{-}{(\mu, p, \texttt{new}\ \ ha) \xrightarrow{\bullet} (\mu, ha; p, \texttt{skip})}$$

Figure 16: Reduction relation for commands

$$\frac{-}{(\cdot, i) \Downarrow \texttt{skip}} \qquad \frac{(p, ch'(v)) \Downarrow c' \qquad ch \neq ch'}{(ch\texttt{(z)}\{c\};\ p, ch'(v)) \Downarrow c'}$$

$$\frac{-}{(ch\texttt{(z)}\{c\};\ p, ch(v)) \Downarrow c[\texttt{z} \mapsto v]}$$

Figure 17: Reduction relation for handler selection

$$\frac{(p, i) \Downarrow c}{(\mu, p) \xrightarrow{i} (\mu, p, i, c)} \qquad \frac{(\mu, p, c) \xrightarrow{o} (\mu', p', c')}{(\mu, p, i, c) \xrightarrow{o} (\mu', p', i, c')}$$

$$\frac{H(ch) = \top}{(\mu, p, ch(v), \texttt{skip}) \xrightarrow{\bullet} (\mu, p)}$$

$$\frac{H(ch) = ch' \qquad (\mu, p) \xrightarrow{ch'(v)} (\mu, p, ch'(v), c)}{(\mu, p, ch(v), \texttt{skip}) \xrightarrow{\bullet} (\mu, p, ch'(v), c)}$$

Figure 18: Reduction relation for programs

disjoint, and $0 \in \mathbb{V}$ as 0 is treated as Boolean false in branching and looping instructions. In our examples we have arithmetic and conditional expressions over $\mathbb{X} \cup \{\texttt{z}\} \cup \mathbb{V}$, with $\mathbb{V} = \mathbb{N}$ and operators defined as usual.

4.2 Semantics

The operational semantics of our language is given as a labeled transition relation on system states, ranged by q. There are two kinds of states. Consumer states denote a system ready to process new input, and are given as a memory-program pair. Producer states denote a system currently handling input, producing output as it goes. Such states are given as a 4-tuple consisting of the current memory, program definition, message being handled, and command being executed in response.

$$q ::= (\mu, p) \mid (\mu, p, i, c)$$

The labeled transition relation on q is defined in terms of the following intermediate judgments.

$(\mu, p, c) \xrightarrow{o} (\mu', p', c')$**:** A small-step labeled reduction stating that, in memory μ, with program p, command c produces o in a single step, modifying μ and p to μ' and p' while doing so, and becoming c'. This reduction relation is given in Figure 16. The only non-standard rules are the `out` $ch(e)$ rule and the `new` ha rule. The former emits output, and the latter adds a handler definition as the head of p.

$(p, i) \Downarrow c$**:** A big-step reduction for handler selection stating that, given program p and input i, c is the command to be executed in response to i. c is the body of the first ch-handler in p, where $i = ch(v)$[11]. In c, any occurrence of the formal parameter `z` has been replaced by v (except those appearing in `new` ha statements). When there is no handler for i in p, command `skip` is chosen. This reduction relation is given in Figure 17.

The labeled transition rules for system states, $q \xrightarrow{s} q'$, are given in Figure 18. The initial state of a reactive system defined by p is the consumer state (μ_0, p). Here, $\mu_0(x) = 0$, for all $x \in \mathbb{X}$. A $q \xrightarrow{i} q'$ transition corresponds to feeding input i to a system in consumer state q (which in turn enters producer state q'). Here, handler selection rules are used to determine which command c to execute in response to i. A $q \xrightarrow{o} q'$ transition corresponds to receiving output from a system in a producer state q (which in turn enters state q'). Output o is the result of taking 1 transition in c, except when $c = \texttt{skip}$, in which case the channel hierarchy H is consulted to check whether the last input channel has a parent. If so, the last input is forwarded to the handler for that parent. If not, the system enters a consumer state. In any case, $\bullet$ is emitted.

4.3 Examples

Program (7), upon receiving $ch_{\text{i}}(v)$, outputs $\bar{ch}_{\text{o}}(5)$ when $v = 0$ and $\bar{ch}_{\text{o}}(4)$ otherwise.

$$ch_{\text{i}}\texttt{(z)}\{\texttt{if z}\,\{\texttt{out}\ \ ch_{\text{o}}\texttt{(}4\texttt{)}\}\,\{\texttt{out}\ \ ch_{\text{o}}\texttt{(}5\texttt{)}\}\} \tag{7}$$

Given $I_1 = [ch_{\text{i}}(0)]$ and $I_2 = [ch_{\text{i}}(1)]$, Program (7) yields $q_0(I_1) = [\bullet, \bar{ch}_{\text{o}}(5), \bullet]$ and $q_0(I_1) = [\ \bullet\ , \bar{ch}_{\text{o}}(4), \bullet]$. Here, q_0 denotes the initial state of the program under consideration. Program (8), upon receiving a message on ch_{i}^2, replace its ch_{i}^1-handler with a handler that, instead of forwarding ch_{i}^1 messages to ch_{o} untouched, adds 1 to the transmitted value.

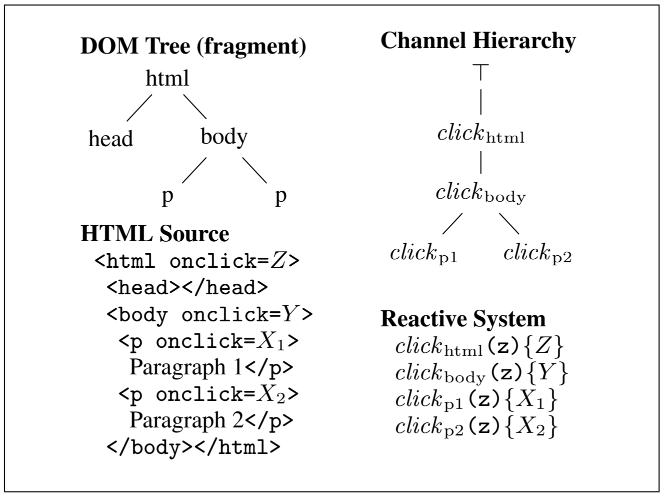
$$\begin{array}{l} ch_{\text{i}}^1\texttt{(z)}\{\texttt{out}\ \ ch_{\text{o}}\texttt{(z)}\} \\ ch_{\text{i}}^2\texttt{(z)}\{\texttt{new}\ \ ch_{\text{i}}^1\texttt{(z)}\{\texttt{out}\ \ ch_{\text{o}}\texttt{(z}+1\texttt{)}\}\} \end{array} \tag{8}$$

Given $I_1 = [ch_{\text{i}}^1(0)]$ and $I_2 = [ch_{\text{i}}^2(5), ch_{\text{i}}^1(0)]$, Program (8) yields $q_0(I_1) = [\bar{ch}_{\text{o}}(0), \bullet]$ and $q_0(I_2) = [\bullet, \bullet, \bar{ch}_{\text{o}}(1), \bullet]$. Program (9) models Program (4) in our language.

```
H(z){h := z}
L(z) {l := 0;
  while 1 {
    out L(l);
    while l = h {skip};
    l := l + 1 } }                (9)
```

Finally, Figure 19 gives an impression of how JavaScript programs can be modeled in our framework. Here, just like in JavaScript

[11] `new` $ch\texttt{(z)}\{c\}$ thus effectively replaces the ch-handler in p.



```
<html onclick=Z>
 <head></head>
 <body onclick=Y>
  <p onclick=X1>
   Paragraph 1</p>
  <p onclick=X2>
   Paragraph 2</p>
 </body></html>
```

$click_{\mathrm{html}}\texttt{(z)}\{Z\}$
$click_{\mathrm{body}}\texttt{(z)}\{Y\}$
$click_{\mathrm{p1}}\texttt{(z)}\{X_1\}$
$click_{\mathrm{p2}}\texttt{(z)}\{X_2\}$

Figure 19: Modeling JavaScript in our Framework

where an `onclick`-event in paragraph 1 causes X_1, Y and Z to be executed in response (in that order (in Firefox)), sending $click_{\mathrm{p1}}(v)$ to the corresponding reactive system will cause X_1 to be executed, whereafter v gets forwarded to the parent handler of $click_{\mathrm{p1}}$ (namely $click_{\mathrm{body}}$), causing Y to be executed, etc.

5. Enforcement

We now develop the static enforcement mechanism for ID-security given in Figures 20 and 21. Along with a program transformation which turns ID-secure programs into IB-secure programs through dynamic enforcement (given in Section 5.1), these two parts form a mechanized approach to rejecting IB-insecure programs[12]. Although the enforcement is phrased as a type system, it is by no means a fundamental choice as there are several viable alternatives such as abstract interpretation [11] for representing the analysis.

Each channel ch has two sub-channels associated with it, one for existence of messages on ch, denoted ch^{e}, and one for content of messages on ch, denoted ch^{c}. If $\mathsf{lbl}(ch) = l_{\mathsf{c}}^{l_{\mathsf{e}}}$, then we set $\mathsf{lbl}(ch^{\mathsf{e}}) = l_{\mathsf{e}}$ and $\mathsf{lbl}(ch^{\mathsf{c}}) = l_{\mathsf{c}}$. The sources (resp. sinks) of our system are the input (resp. output) sub-channels. When analyzing information flow in p, we are interested in knowing how p relates sources and sinks. We (over)approximate this relationship with a mapping Γ. $\Gamma(ch^{\mathsf{e}})$, resp. $\Gamma(ch^{\mathsf{c}})$, is the set of sources that an observer capable of observing existence, resp. content, of messages on ch can obtain information from (by observing presence of messages, resp. values passed, on ch). $\Gamma(ch^{\mathsf{e}}) \subseteq \Gamma(ch^{\mathsf{c}})$, for all ch, since being capable of observing values on ch necessarily implies being capable of observing that *some* message was sent on ch. The type checker checks whether a Γ correctly (over)approximates information flows in p, in which case p has type Γ, written $\vdash p : \Gamma$.

We let $C^{\mathsf{e}} = \{ch^{\mathsf{e}} \mid ch \in C\}$ for any $C \subseteq \mathbb{C}$. Likewise for C^{c}. Then $\Gamma : \mathbb{I} \to \mathcal{P}(\mathbb{I})$, where $\mathbb{I} = \mathbb{C}^{\mathsf{e}} \cup \mathbb{C}^{\mathsf{c}}$ is the set of sources and sinks, ranged by a. The set of sink types is the powerset of sources. These form a lattice, with $\sqsubseteq$, $\sqcap$ and $\sqcup$ defined as $\subseteq$, $\cap$ and $\cup$. In this way, our enforcement mechanism resembles the flow-sensitive security-type system of [24]. There the powerset of information sources is a "universal" flow lattice $\mathcal{L}_{\mathrm{U}}$ which all other flow lattices $\mathcal{L}'$ can be defined in terms of, and that a principal type of $\mathcal{L}'$ can be derived from the principal type of $\mathcal{L}_{\mathrm{U}}$. $\Gamma \sqsubseteq \Gamma'$ if $\Gamma(a) \sqsubseteq \Gamma'(a)$, $\forall a \in \mathbb{I}$, so the Γs themselves form a lattice. Any typable p thus has a principal (that is, least) type.

[12] You can in fact replace our type system with any sound enforcement of ID-security.

$$\frac{pc_1 \vdash \Gamma_1\ \mathsf{p}\ \{c\}\ \mathsf{p}'\ \Gamma_1'}{pc_2 \vdash \Gamma_2\ \mathsf{p}\ \{c\}\ \mathsf{p}'\ \Gamma_2'}\, p_2 \sqsubseteq p_1,\ \Gamma_2 \sqsubseteq \Gamma_1,\ \Gamma_1' \sqsubseteq \Gamma_2'$$

$$\frac{-}{pc \vdash \Gamma\ \mathsf{p}\ \{\texttt{skip}\}\ \mathsf{p}\ \Gamma} \qquad \frac{-}{pc \vdash \Gamma\ \mathsf{p}\ \{ha\}\ pc : ha;\ \mathsf{p}\ \Gamma}$$

$$\frac{pc \vdash \Gamma\ \mathsf{p}\ \{c_1\}\ \mathsf{p}'\ \Gamma' \qquad pc \vdash \Gamma'\ \mathsf{p}'\ \{c_2\}\ \mathsf{p}''\ \Gamma''}{pc \vdash \Gamma\ \mathsf{p}\ \{c_1\,\texttt{;}\ c_2\}\ \mathsf{p}''\ \Gamma''}$$

$$\frac{\Gamma \vdash e : T}{pc \vdash \Gamma\ \mathsf{p}\ \{x\ \texttt{:=}\ e\}\ \mathsf{p}\ \Gamma[x \mapsto T \sqcup pc]}$$

$$\frac{\Gamma \vdash e : T}{pc \vdash \Gamma\ \mathsf{p}\ \{\texttt{out}\ \ ch\texttt{(}e\texttt{)}\}\ \mathsf{p}\ \Gamma[ch \mapsto \langle T, pc\rangle]}$$

$$\frac{\Gamma \vdash e : pc' \qquad pc \sqcup pc' \vdash \Gamma\ \mathsf{p}_{i-1}\ \{c_i\}\ \mathsf{p}_i\ \Gamma' \qquad i = 1,2}{pc \vdash \Gamma\ \mathsf{p}_0\ \{\texttt{if}\ \ e\ \{c_1\}\ \{c_2\}\}\ \mathsf{p}_2\ \Gamma'}$$

$$\frac{\Gamma \vdash e : pc' \qquad pc \sqcup pc' \vdash \Gamma\ \mathsf{p}\ \{c\}\ \mathsf{p}'\ \Gamma}{pc \vdash \Gamma\ \mathsf{p}\ \{\texttt{while}\ \ e\ \{c\}\}\ \mathsf{p}'\ \Gamma}$$

Figure 20: Command Type Rules

$$\frac{-}{\vdash \cdot : \Gamma} \qquad \frac{\vdash \bot : ha;\ p : \Gamma}{\vdash ha;\ p : \Gamma}$$

$$\frac{pc \sqcup \check{ch}^{\mathsf{e}} \vdash \Gamma[\mathtt{z} \mapsto \check{ch}^{\mathsf{c}}]\ \mathsf{p}\ \{c\}\ \mathsf{p}'\ \Gamma[\mathtt{z} \mapsto \check{ch}^{\mathsf{c}}] \qquad \Gamma \vdash \mathsf{p}'}{\vdash (pc : ch\texttt{(z)}\{c\};\ \mathsf{p}) : \Gamma}$$

Figure 21: Handler Type Rules

The type system *assumes* that loops and handlers can run an arbitrary number of times, handlers can be run in any order, and any possible definition of a handler is considered possibly active at any time. Therefore, when a `new` command is encountered during typing, it is brought to the "top level" (in a sense "flattening" p), and typed there in the pc the `new` command was discovered in. So we are in fact type checking a (slightly) richer syntactic category, $\mathsf{p} ::= p \mid pc : ha; \mathsf{p}$, of programs where handlers can be paired with the pc they were discovered in. Notice that this (simplified) version of our type system infers nothing. It requires Γs of the form $\Gamma : \mathbb{I} \cup \mathbb{X} \to \mathcal{P}(\mathbb{I})$. We note, though, that a principal $\Gamma : \mathbb{I} \to \mathcal{P}(\mathbb{I})$ *can* indeed be inferred from p. This inference involves several fixed point computations to make sure the inferred Γ is (over)approximative wrt. the above assumptions.

Let T range over $\mathcal{P}(\mathbb{I})$. Here, $\Gamma[x \mapsto T]$ *replaces* the set of sources Γ says can leak into x, with T. This makes our type system *flow-sensitive*, taking into account the order of command execution. This is in sharp contrast to flow-insensitive type systems, such as those of [7, 54], which over-approximate by assigning the same type to $l\ \texttt{:=}\ h\texttt{;}\ \texttt{out}\ \ L\texttt{(}l\texttt{)}\texttt{;}\ l\ \texttt{:=}\ 0$ and $l\ \texttt{:=}\ h\texttt{;}\ l\ \texttt{:=}\ 0\texttt{;}\ \texttt{out}\ \ L\texttt{(}l\texttt{)}$, for instance. However, $\Gamma[ch \mapsto \langle T, T'\rangle]$ *inserts* the content of $T \cup T'$ (resp. T') to the set of sources Γ says can leak into ch^{c} (resp. ch^{e}).

$$\Gamma[x \mapsto T](x') = \begin{cases} T & \text{if } x = x', \\ \Gamma(x') & \text{otherwise.} \end{cases}$$

$$\Gamma[ch \mapsto \langle T, T'\rangle](y) = \begin{cases} \Gamma(y) \sqcup T' & \text{if } ch^{\mathsf{e}} = y, \\ \Gamma(y) \sqcup T \sqcup T' & \text{if } ch^{\mathsf{c}} = y, \\ \Gamma(y) & \text{otherwise.} \end{cases}$$

We let $\check{ch} = \{ch' \in \mathbb{C} \mid \exists n \in \mathbb{N}\,.\, H^n(ch') = ch\}$, that is, the set of all descendants of ch. c can be run as a reaction to receiving a message on any $ch' \in \check{ch}$, so c runs in the context containing information about the existence of messages on all channels in $\check{ch}$. $\mathtt{z}$ could contain content from any of the $\check{ch}$ channels. We assume the presence of a typing relation for expressions $\Gamma \vdash e : T$, with the requirement that for each variable x in e, $\Gamma(x) \sqsubseteq T$.

The command typing judgment $pc \vdash \Gamma\ \mathsf{p}\ \{c\}\ \mathsf{p}'\ \Gamma'$ should be read "under context pc, command c takes an initial flow approximation Γ and program p to Γ' and program p'". Most of the rules are standard, save the first rule in Figure 20, referred to as the "weakening rule". While the other rules permissively approximate information flow in c, this rule allows us to conclude that more flows occur in c (yielding a weaker guarantee). This is needed when typing `while` commands and whole handler bodies, as the typing must approximate an arbitrary number of executions of these. The program typing judgment $\vdash p : \Gamma$ should be read "Γ (over)approximates information flows in p". The only interesting rule in Figure 21 is the last one, which types a handler together with the pc it was discovered in while traversing p. It types the handler body c under context $pc \sqcup \check{ch}^{\mathsf{e}}$, where $\check{ch}^{\mathsf{e}}$ is the information conveyed by the existence of a message on ch, and any of its subchannels.

We close this section with the type soundness theorem. Γ is *consistent with the channel labeling* if $\forall a \in \mathsf{dom}(\Gamma).\forall a' \in \Gamma(a).\mathsf{lbl}(a') \sqsubseteq \mathsf{lbl}(a)$. Also, p is well-typed if $\vdash p : \Gamma$ for some consistent Γ. At last,

Theorem 5.1. *If p is well-typed, then p is ID-secure.*

5.1 Buffering Output

Ideally, a reactive system should stay reactive. Thus, one would usually expect an event handler to always terminate, yielding finite output, and allowing the reactive system to process the next input symbol. For instance, in JavaScript, `timeOut` events are handled with a lower priority than other events to prevent procedurally-created events from starving other events. Also, when a JavaScript program enters an infinite loop, the browser asks the user whether he wants to terminate the reaction prematurely. One could argue that any diverging program is either the product of a programming error or a programmer with malicious intent, making the program diverge in the hope that doing so makes the program pass a static enforcement check and still leak.

We present an encoding of programs which makes a program buffer its output until it is ready to process a new input. Buffering output mitigates the bandwidth of leaks due to intermediate output. One downside is that this encoding will mute handlers that diverge while producing output. However, our justification for considering buffering useful is that programs with diverging handlers do not belong to the paradigm of reactive systems. The program transformation $\mathsf{buff}(p)$, given in Figure 22, replaces each output command $\mathtt{out}\ \ ch(v)$ with an "enqueue" command enq, queuing $\mathsf{idx}(ch)$ and v in q. Here, idx is a bijection from the n channels occurring in p and $\{1..n\}$. Control for flushing the queue, c_{o}, is then added at the end of each root handler. deq yields the next element in a queue and $drop$ drops the next element in the queue. c_{i} initializes the queue q to the empty queue if q has the initial variable value 0.

The effect of this buffering is the same as that of running the original program in a wrapper which buffers outputs, like the one given in Figure 14. This leads to the following observation.

Theorem 5.2. *If p is ID-secure, then* $\mathsf{buff}(p)$ *is IB-secure.*

We summarize the quantitative implication in this theorem.

Theorem 5.3. *If* $\vdash p$*, then* $\mathsf{buff}(p)$ *is* $\log_2(n+1)$*-bit secure where* n *is the number of observables in the input to* $\mathsf{buff}(p)$.

So, applying a JavaScript implementation of $\mathsf{buff}(\cdot)$ on the script from Figure 1 yields an IB-secure program, thus limiting the bandwidth of the progress channel from arbitrary to $\log_2(n+1)$.

$\mathsf{buff}(\cdot)$ is homomorphic for recursively defined objects, and leaves atomic objects unchanged, with these exceptions:

$$\mathsf{buff}(ch(\mathtt{z})\{c\}) = \begin{cases} ch(\mathtt{z})\{c_{\mathrm{i}};\ \mathsf{buff}(c);\ c_{\mathrm{o}}\} \\ \quad \text{if } H(ch) = \top, \\ ch(\mathtt{z})\{c_{\mathrm{i}};\ \mathsf{buff}(c)\} \\ \quad \text{otherwise.} \end{cases}$$
$$\mathsf{buff}(\mathtt{out}\ \ ch(e)) = q \mathrel{\mathtt{:=}} enq\ \mathsf{idx}(ch)\ enq\ e\ q$$

Here,

```
c_i = if  q = 0 {q := emptyq} {skip}
c_o = while  q ≠ emptyq {
        ch_out := deq q;
        val := deq drop q;
        q := drop drop q;
        if  ch_out = 1 {out  ch_1(val)} {skip};
        ⋮
        if  ch_out = n {out  ch_n(val)} {skip}}
```

Figure 22: Buffering Encoding

6. Related work

Security of event-driven systems has been investigated in the context of process calculi [17, 20, 21, 25, 39, 44, 45] and event-based abstractions [31, 32, 46]. Connections with security models for more concrete programming languages have been made [18, 33]. However, relatively little has been done on exploring the flow of information through language constructs in reactive languages.

Sabelfeld and Mantel [46] investigate the impact of different types of channels (secret, encrypted, public) and different types of communication (synchronous and asynchronous) on information-flow security. The encrypted channel is similar to our existential channel, where only the presence (not the content) of messages is visible to attackers. The origins of existence and content levels are in security labels for datatypes. For example, Jif [36, 37] allows arrays, where the length of the array is public but the individual elements are secret.

O'Neil et al. [38] investigate the security of interactive programs. They focus on protecting secret user strategies from leaking to the adversary. Clark and Hunt [9] note that it makes no difference in a deterministic setting whether the input/output is represented by strategies or streams. As discussed in Section 1, ONeil et al. [38], as well as Askarov and Sabelfeld [3], consider termination-sensitive noninterference. The price of termination-sensitivity is restrictiveness: loops with secret guards will likely break security and will hence be rejected by the respective enforcements.

Almeida Matos et al. [34] propose a type system for noninterference and nondisclosure properties. They focus on suspension features and leaks associated with them. Communication is modeled by streams in security formalizations by Askarov et al. [1] for a language with cryptographic primitives and by Askarov and Sabelfeld [3] for a language with dynamic code evaluation and declassification primitives.

Askarov et al. [2] clarify the impact of leaking information via (non)termination of programs in the presence of intermediate output. Restrictions on language constructs that might result in abnormal termination or divergence, originating in classical security analysis [13, 54] and supported in modern information-flow tools Jif [37], FlowCaml [50], and the SPARK Examiner [6, 8], are not strong enough to prevent brute-force attacks as Program 4.

As mentioned in Section 2, Bohannon et al. propose security definitions for reactive systems that correspond to four indistinguishability relations on streams. They emphasize (progress-sensitive) CP-security and (progress-insensitive) ID-security and choose to focus on the latter. Distinct feature of our approach compared to that of Bohannon et al. is (i) simple framework (finite inductive streams rather than infinite streams and coinductive definitions), (ii) new handler creation, (iii) strong security guarantees (the security definition of Bohannon et al. is similar in spirit to *PINI* [2] which allows leaking secrets entirely via the intermediate output, whereas we allow only one bit to be leaked at most per consumed public input), (iv) distinguishing the security level of message existence and content, (v) output buffering to guarantee strong security, and (vi) a more permissive flow-sensitive enforcement.

Askarov et al. [2] demonstrate that progress-insensitive noninterference allows leaking secrets in non-polynomial time in the size of the secret. In contrast, our security condition provides a tight quantitative guarantee: the number of leaked bits is bounded by $\log_2(n+1)$, where n is the number of public inputs. Quantitative information-flow security is a mature area by itself. Smith [51] provides an excellent summary of the state of the art. We adopt Smith's min-entropy based definition of quantitative security in our paper. To the best of our knowledge, quantitative security of reactive programs has not been explored previously.

Devriese and Piessens [14] suggest splitting the execution of a program onto threads operating at different security levels. Only the thread at a given level is allowed to consume input from a channel labeled with level. A similar mechanism is in place for output.

Tracking information flow in web applications is becoming increasingly important, e.g., recent highlights are a server-side mechanism by Huang et al. [23] and a client-side mechanism for JavaScript by Vogt et al. [52], although, like a number of related approaches, they do not discuss soundness. Mozilla's ongoing project FlowSafe [15] aims at extending Firefox with runtime information-flow tracking, where dynamic information-flow monitoring [4, 5] lies at its core. Recently, Magazinius et al. [30] have proposed how to support decentralized policies with possible mutual distrust for dynamically tracking information flow in mashups.

7. Conclusion

We have proposed a framework for information-flow security of reactive programs. The framework tightly regulates the bandwidth of leaks due to intermediate output: at most $\log(n+1)$ bits are allowed to be released, where n is the number of public inputs to the program. This provides much-desired middle ground between the Draconian progress-sensitive and the brute-force attackable progress-insensitive security. The framework includes a flexible treatment of channels: it is possible to reveal the existence of messages and at the same time protect their content. We address features of reactive programs that are important in a dynamic environment (such as in a web browser): new handler creation and hierarchical event handling. Although our security requirement is strong, it is realizable: we have presented a combination of flow-sensitive static analysis and output buffering to guarantee security. The model scales up to handle exceptions due to the insensitivity to abnormal termination can be treated in the same way as nontermination. Thus, uncaught exceptions due to, say, partial operators, in high context correspond to looping in high context which is allowed by both our enforcement and security condition.

Future work includes explorations of further features of reactive languages, which will allow us to treat channels as first-class values. Another important direction of current and future work is integration of our approach with the larger research program [3, 30, 41, 43] and experiments with case studies. Of particular focus is supporting policies for intentional information release or *declassification* [3] (including decentralized policies such as in web mashups [30]), timeout events [41], and interaction with the DOM tree [43]. We are experimenting with an enforcement mechanism for JavaScript that is based on an inlining transformation.

In a malicious-code scenario, it is important to cover all possible channels of leaking information. This paper gives particular attention to the leaks via intermediate output because they can be magnified into brute-force attacks, as illustrated in the example in Section 1. Other information channels such as via timing [41] and resource exhaustion [2] are important directions of future work.

We are investigating dynamic enforcement by runtime monitoring along the lines of recent series of work on dynamic information-flow tracking [3–5, 26, 42, 48, 49]. Dynamic enforcement provides immediate advantages for handling dynamic language constructs and extending our approach to dynamic channel hierarchies.

We anticipate it is straightforward to generalize our security framework to state-transition systems and parametrize on when buffering is done. We expect a generalization of Theorem 5.3 to guarantee that high-bandwidth leaks via progress of single events are mitigated into low-bandwidth leaks via progress of event chunks. However, as the focus of the present paper is on reactive systems, such a framework is subject to future investigation.

Finally, we are exploring the possibility of giving the programmer control over flushing the output buffer. When several public inputs can be processed until the output buffer is flushed, we have the potential of providing stronger guarantees on the number of leaked bits. The potential of this alternative depends on common usage patterns in existing applications, which we plan to roadmap.

Acknowledgments Thanks are due to Cédric Fournet for the suggestion of buffering output and to Daniel Hedin, Sebastian Hunt and anonymous reviewers for useful discussions. This work was funded by the European Community under the WebSand project and the Swedish research agencies SSF and VR.

References

[1] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. *Theoretical Computer Science*, 402:82–101, August 2008.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, October 2008.

[3] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[4] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[5] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.

[6] J. Barnes and JG Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[7] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security*, pages 79–90, November 2009.

[8] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, 24(4):39–46, 2004.

[9] D. Clark and S. Hunt. Noninterference for deterministic interactive programs. In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, October 2008.

[10] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approxima-

tion of fixpoints. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 238–252, January 1977.

[12] D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.

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

[14] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy*, May 2010.

[15] B. Eich. Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, October 2009.

[16] Facebook. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, 2009.

[17] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *J. Computer Security*, 3(1):5–33, 1995.

[18] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging language-based and process calculi security. In *Proc. Foundations of Software Science and Computation Structure*, volume 3441 of *LNCS*, pages 299–315. Springer-Verlag, April 2005.

[19] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[20] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. European Symp. on Programming*, volume 1782 of *LNCS*, pages 180–199. Springer-Verlag, 2000.

[21] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 81–92, January 2002.

[22] Arnaud Le Hors and Philippe Le Hegaret. Document Object Model Level 3 Core Specification. Technical report, The World Wide Web Consortium, 2004.

[23] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. International Conference on World Wide Web*, pages 40–52, May 2004.

[24] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 79–90, 2006.

[25] N. Kobayashi. Type-based information flow analysis for the pi-calculus. Technical Report TR03-0007, Tokyo Institute of Technology, October 2003.

[26] G. Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.

[27] G. Lowe. Quantifying information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 18–31, June 2002.

[28] S. Maffeis, J.C. Mitchell, and A. Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proc. of ESORICS'09*. LNCS, 2009.

[29] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[30] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, April 2010.

[31] H. Mantel. Possibilistic definitions of security – An assembly kit –. In *Proc. IEEE Computer Security Foundations Workshop*, pages 185–199, July 2000.

[32] H. Mantel. Information flow control and applications—Bridging a gap. In *Proc. Formal Methods Europe*, volume 2021 of *LNCS*, pages 153–172. Springer-Verlag, March 2001.

[33] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Computer Security*, 11(4):615–676, September 2003.

[34] A. Almeida Matos, G. Boudol, and I. Castellani. Typing non-interference for reactive programs. *Journal of Logic and Algebraic Programming*, 72:124–156, 2007.

[35] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, 2008.

[36] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.

[37] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001.

[38] K. O'Neill, M. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 190–201, July 2006.

[39] F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proc. IEEE Computer Security Foundations Workshop*, pages 320–330, June 2002.

[40] W. Rafnsson and A. Sabelfeld. Limiting information leakage in event-based communication: Extended version. Technical report, Chalmers University of Technology, 2011. Located at `http://www.cse.chalmers.se/~rafnsson/2011plas`.

[41] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[42] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

[43] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proc. European Symp. on Research in Computer Security*, LNCS. Springer-Verlag, September 2009.

[44] P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 1–62. Springer-Verlag, 2001.

[45] P. Ryan and S. Schneider. Process algebra and non-interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 214–227, June 1999.

[46] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, September 2002.

[47] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[48] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[49] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE Computer Security Foundations Symposium*, pages 203–217, July 2007.

[50] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.

[51] G. Smith. On the foundations of quantitative information flow. In *Proc. Foundations of Software Science and Computation Structure*, volume 5504 of *LNCS*, pages 288–302, March 2009.

[52] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, February 2007.

[53] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

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