Static Provenance Verification for Message Passing Programs

Rupak Majumdar[1], Roland Meyer[2], and Zilong Wang[1]

[1] MPI-SWS, Germany
[2] University of Kaiserslautern, Germany

Abstract. Provenance information records the source and ownership history of an object. We study the problem of provenance tracking in concurrent programs, in which several principals execute concurrent processes and exchange messages over unbounded but unordered channels. The provenance of a message, roughly, is a function of the sequence of principals that have transmitted the message in the past. The provenance verification problem is to statically decide, given a message passing program and a set of allowed provenances, whether the provenance of all messages in all possible program executions, belongs to the allowed set. We formalize the provenance verification problem abstractly in terms of well-structured provenance domains, and show a general decidability result for it. In particular, we show that if the provenance of a message is a sequence of principals who have sent the message, and a provenance query asks if the provenance lies in a regular set, the problem is decidable and EXPSPACE-complete.
While the theoretical complexity is high, we show an implementation of our technique that performs efficiently on a set of Javascript examples tracking provenances in Firefox extensions. Our experiments show that many browser extensions store and transmit user information although the user sets the browser to the private mode.

1 Introduction

Controlled access and dissemination of data is a key ingredient of system security: we do not want secret information to reach untrusted principals and we do not want to receive bad information (indirectly) from untrusted principals. Many organizations receive private information from users and this information is passed around within the organization to carry out business-critical activities. These organizations must ensure that the data is not accidentally disclosed to unauthorized users, as the potential cost of disclosure can be high. Moreover, in many domains, such as healthcare and finance, the control of data is required by regulatory agencies through legislation such as HIPAA and GLBA.

We present an abstract model of information dissemination in message passing systems, and a static analyzer to verify correct dissemination. We model systems as concurrent message passing processes, one process for each principal in the system. Processes communicate by sending and receiving messages via a shared set of channels. Channels are unbounded, but can reorder messages. Sends are non-blocking, but receive actions block until a message is available.

To track information about the origin and access history of a message, we augment messages with *provenance* annotations. Roughly, the provenance of a message is a function of the sequence of principles that have transmitted the message in the past. Depending on the function, we get different provenance annotations. For example, the annotation can simply be the sequence of principals. Whenever a principal sends a message, we append the name of the principal to the current provenance of the message. The *provenance verification problem* asks, given a message passing program, a variable in the program, and a set of allowed provenance annotations, whether the provenance of every message stored in the variable, on every run of the program, belongs to the set of allowed provenances.

Consider a healthcare system in which a patient sends health questions to a secretary or a nurse, who in turn, forwards the question to doctors. An information-dissemination policy may require that every health answer received by the patient has been seen by at least one doctor. That is, the provenance of every message received by the patient must belong to the regular language $\mathsf{Patient}(\mathsf{Secretary} + \mathsf{Nurse})\ \mathsf{Doctor}^+$.

We consider provenance verification for general provenance domains satisfying an algebraic requirement. Static provenance verification is hard because of two sources of unboundedness in the model. First, the provenance information associated with a single message can be unbounded. For example there is no bound on the number of doctors who see a health question before an answer is sent back. Second, the number of pending messages in the system can be unbounded. We tackle these two sources of unboundedness as follows.

We give a reduction from provenance verification problem to coverability in *labeled* Petri nets, where tokens carry (potentially unbounded) provenance data. As a result, we obtain a general decidability result for provenance verification problem, when the domain of provenance annotations is well-structured [1,8]. Specifically, we show verification is EXPSPACE-complete for the set provenance domain, that tracks the set of principals that have seen a message, as well as for the language provenance domain, in which provenance information is stored as ordered sequences of principals that have seen the message and policies are regular languages. Our proofs combine well-structuredness arguments with symbolic representations; we analyze coverability in a product of a Petri net modeling the system and a symbolic domain encoding the set of allowed provenances.

While our decision procedures reduce the verification problems to problems on Petri nets, our experiences with a direct implementation of provenance verification based on existing Petri net coverability tools have been somewhat disappointing. Mostly, this is because after the reduction to Petri nets, the coverability tools fail to utilize the structure of message passing programs, in particular potential state-space reductions arising from partial order reduction (POR) [11].

We implemented a coverability checker that is tuned for message passing programs on top of the Spin model checker [14]. Our implementation uses the expand-enlarge-check (EEC) pradigm [10]. The EEC algorithm explores a sequence of finite-state approximations of the message passing program. Intu-

itively, the approximation is obtained by replacing the counters in the Petri net with "abstract" counters that count precisely up to a given parameter k, and then set the count to ∞. Since the induced state space is finite for each approximation, we can use a finite-state reachability engine (such as Spin) to explore its state space. Additionally, we use partial order reduction, already implemented in Spin, to reduce the explored state space, allowing local actions of different processes to commute.

Our choice of a message passing programming model with unbounded but unordered buffers was inspired by the communication model in browser extensions, where several components communicate asynchronously. Specifically, we checked the following property of extensions. Most browsers have a "private mode" that allow users to browse the internet without saving information about pages visited. Browser extensions should respect the private mode and not save user information (or worse, upload user information to remote servers) while the user is browsing in the private mode. We checked this property and found that several widely-used Firefox extensions, including some extensions whose purpose is to improve user privacy, do not properly handle "private mode" settings. Among nine browser extensions using message passing, local storage, and sometimes remote database accesses, we found five extensions store user data even in the private mode. Thus, our experiments demonstrate that a precise static tool can be useful in detecting privacy violations in this domain.

One can view our result as a general compilation procedure from a provenance verification problem for a program P to a safety verification problem for an instrumented program P'. The instrumentation P' adds some counters to P but keeps the other features (e.g., complex control flow and data structures) the same: program P' is safe iff P satisfies the provenance properties. After the reduction, we can harness any verification technique that has been developed for the underlying class of programs (e.g., abstract interpretation or software model checking). Our experiments use a simple dataflow abstraction, but other abstract domains could be used for more precision. We chose message passing programs for our presentation as they capture the essence of provenance tracking: concurrency, unbounded provenance information, and unbounded channels. This focus allows us to settle the complexity of provenance verification without mixing it with the complexity of features in the programming model.

Related Work Provenance annotation on data has been studied extensively in the database community [6,3,12], both for annotating query results and for tracking information through workflows. Provenance information is usually tracked for a fixed database and a fixed query in a declarative query language. Seen as a program, the query has exactly one "execution path." The connection between provenance tracking and dependency analysis in (sequential) programs was made in [5]. A provenance-tracking semantics for asynchronous π-calculus was given in [27], but the static analysis problem was not considered. Most previous work focused on dynamic tracking and enforcement along one execution path, and the static meet-over-all-paths solution was not considered. In contrast, we provide algorithms to track provenances in concurrent message passing programs, and

```
patient {
  var p1, p2, p3;
  while (true) {
    choose
P1: [] p1 = HQ; send(ch0, p1);
P2: [] p1 = AR; send(ch0, p1);
P3: [] recv(ch1, p2);
P4:    if (p2 == HA) p3 = p2;
  }
}
```

```
doctor{
  var d1, d2;
  while(true) {
D1: recv(ch2, d1);
D2: d2 = HA(d1);
D3: choose
D4: [] send(ch1, d2);
D5: [] send(ch2, d2);
  }
}
```

```
secretary {
  var s1, s2;
  while (true) {
S1: recv(ch0, s1);
S2: if (s1 == HQ)
S3:   send(ch2, s1);
S4: else {
S5:   s2 = AA(s1);
S6:   send(ch1, s2);
    }
  }
}
```

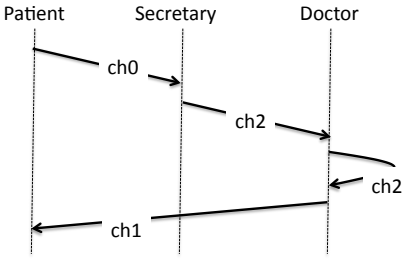

Fig. 1. Medical system example

give algorithms to check provenance queries over all execution paths of programs. We were inspired by the algebraic framework of provenance semirings [12] to give a similar algebraic description of provenance domains.

Our algorithm for provenance verification generalizes algorithms for explicit information flow studied in the context of sequential programs [25], e.g., through taint analysis. Taint analysis problems [15,19] classify methods as *sources*, *sinks*, and *sanitizers*, and require that any data flow from sources to sinks must go through one or more sanitizers. In our model, this property can be formulated by requiring that the provenance of every message received by a sink must conform to the regular specification $(source^+\ sanitizer^+)^*$. We are able to verify such properties for message passing programs, where the source, sanitizer, and sink can be concurrently executing processes sharing unbounded channels, and with other intermediary processes as well. Previous work, too numerous to enumerate here, either dealt with dynamic enforcement or provided imprecise static checks for these domains. We show *precise* static analysis remains decidable!

2 Example

We motivate our results by modeling a simple online health system described in [2], which allows patients to interact with their doctors and other healthcare professionals using a web-based message passing system. In the system, users have different roles, such as Patient, Secretary, and Doctor. Patients can ask health questions and receive answers by exchanging messages with their doctors.

For simplicity of exposition, we describe a subset of the functionality of the system as a message passing program. (In Section 5, we modeled the entire system as a case study.) Intuitively, a message passing program is a collection of imperative processes running concurrently, one for each principal in the system. In our example, each role (Patient, Doctor, etc.) is modeled as a different principal. The processes run by the principals have local variables, and in addition, communicate with each other by sending to and receiving from shared channels. We assume shared channels are potentially unbounded, but may reorder messages. Message sends are non-blocking, the execution continues at the control point following the send. Receives are blocking: a process blocks until some message from the channel is received.

Figure 1 shows a simple implementation of the system, written in a simple imperative language. We have three principals: Patient (modeling the set of patients using the system), Secretary (modeling secretaries who receive and forward messages), and Doctor (modeling the set of doctors using the system). The `choose` construct nondeterministically chooses and executes one of its branches. A `send` action sends a message to a channel, and a `recv` receives a message from a channel into a local variable.

There are four kinds of messages in the system. The patient can send a health question (HQ) or an appointment request (AR). The healthcare providers can send back a health answer (HA) or an appointment confirmation (AA). The principals communicate through shared channels `ch0`, `ch1`, and `ch2`.

The patient process runs in a loop. In each step, it nondeterministically decides to either send an HQ or an AR to `ch0`, or to receive an answer on channel `ch1`. The secretary process runs a loop. In each step, it receives a message from channel `ch0`. If it is an HQ, the message is forwarded to doctors on channel `ch2`. If it is an AR, the secretary answers the patient directly on channel `ch1`. The doctor process receives health questions on channel `ch2`. It computes a health answer based on the received message (the assignment on line `D2`). It can either reply directly to the patient (on channel `ch1`), or put the answer back to channel `ch2` for further processing.

Figure 1 also shows a possible message sequence for a health question, where the patient sends a health question to the secretary, the secretary forwards it to the doctor, and the doctor looks at the message several times before replying with a health answer. We capture the flow of messages through the principals using provenance annotations with each message; the provenance captures the history of all the principals that have forwarded the message. While in Section 3 we give a general algebraic definition of a *provenance domain*, for the moment, think of a provenance as a string over the principals. When a message is initially assigned, e.g., on line `P1`, the provenance is the empty string ε. After the patient sends the message, the channel `ch0` contains an HQ message with provenance Patient. When the message is forwarded to channel `ch2`, its provenance becomes Patient Secretary. Finally, when the message is sent back on `ch1`, its provenance is a string in the regular language Patient Secretary Doctor$^+$, indicating that it

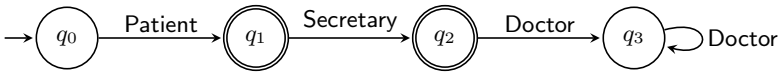

Fig. 2. Complemented finite automaton for provenance property. We omit an accepting sink to which all unspecified edges go.

has been sent originally by the patient, seen by the secretary next, and then seen by the doctor one or more times.

The *provenance verification problem* asks, given the message passing program, a variable `v`, and a regular language R of provenances, whether the content of `v` has a provenance in R along all program executions. In the example, we can ask if the provenance of variable `p3` is in the set

$$\varepsilon + \mathsf{Patient}\ \mathsf{Secretary}\ \mathsf{Doctor}^{+}, \tag{1}$$

capturing the requirement that any health answer must be initiated by a health question from the patient, and must be seen by a doctor at least once, after it has been seen by a secretary.

Notice that the example is unbounded in two dimensions. First, the channels can contain unboundedly many messages. For example, the patient process can send unboundedly many messages on channel `ch0` before the secretary process receives them. Second, the provenance annotations can be unbounded: a message in channel `ch2` can have an unbounded number of Doctor annotations.

We show the provenance verification problem is decidable. The first observation is that, if we ignore provenances, we can keep a counter for each channel `ch` and each message type m, that counts the number of messages with value m that are currently in ch. A send action increases the counter, a receive decrements it. We can then show that the transition system of a message passing program is *well-structured* [1,8]: an action that could be taken in a state can also be taken if there are more messages in the channels. Formally, we give a reduction to Petri nets, an infinite-state well-structured system with good decidability properties.

In the presence of provenances, we have to be more careful. Unlike a normal Petri net, now the "tokens" (the messages in the channels) will carry potentially unbounded provenance annotations. However, given the regular set R, we only need to distinguish two provenance annotations that behave differently with respect to a deterministic finite automaton A for R. So, we keep more counters that are now of the form $\langle \mathtt{ch}, m, q\rangle$: one counter for each combination of channel `ch`, message type m, and state q of A. The state of the automaton A remembers where the automaton would go to, starting with its initial state, on seeing the provenance annotation. Similarly, for each variable in the program, we distinguish the contents of the variable based on the message type m as well as the state q of the automaton.

Figure 2 shows a deterministic automaton accepting the complement of the language in (1). Using this automaton, we describe the reduction to a well-structured system as follows. Let $Q = \{q_0, q_1, q_2, q_3, q_4\}$ be the set of states of the automaton (q_4 is the omitted sink state). We have a set of integer-valued counters $\langle \mathtt{ch}i, m, q\rangle$, for $i = 0, 1, 2$, $m \in \{\mathtt{HQ}, \mathtt{HA}, \mathtt{AA}, \mathtt{AR}\}$, and $q \in Q$. For example,

```
patient {
  var p1, p2, p3;
  while (true) {
    choose
P1′  [] p1 = ⟨HQ,q0⟩; ⟨ch0, HQ,q1⟩++;
P2′  [] p1 = ⟨AR,q0⟩; ⟨ch0, AR,q1⟩++;
P3_1 [] if ⟨ch1,HQ,q⟩ > 0 (for each q ∈ Q)
        p2 = ⟨HQ,q⟩; ⟨ch1,HQ,q⟩--;
P3_2 [] if ⟨ch1,HA,q⟩ > 0 (for each q ∈ Q)
        p2 = ⟨HA,q⟩; ⟨ch1,HA,q⟩--;
P4′     if (p2 == (HA, ·) ) p3 = p2;
P3_3 [] if ⟨ch1,AA,q⟩ > 0 (for each q ∈ Q)
        p2 = ⟨AA,q⟩; ⟨ch1,AA,q⟩--;
P3_4 [] if ⟨ch1,AR,q⟩ > 0 (for each q ∈ Q)
        p2 = ⟨AR,q⟩; ⟨ch1,AR,q⟩--;
  }
}
```

Fig. 3. Translation of `patient`. We have simplified some statements for readability: the actual translation performs a case split over `p1` in lines `P1′` and `P2′`, and performs the check on line `P4′` after each statement $\texttt{P3}_i$.

the counter $\langle \texttt{ch0}, \texttt{HQ}, q_1 \rangle$ stores the number of `HQ`s in `ch0` for which the automaton is in state q_1. Figure 3 shows the translation of the `patient` process. The send actions are replaced by incrementing the appropriate counter. For example, the action `send(ch0,p1)` in line `P1` is replaced with incrementing the counter $\langle \texttt{ch0}, \texttt{HQ}, q_1 \rangle$, the state of the automaton is q_1 because the principal Patient takes the automaton from its initial state q_0 to the state q_1. The receive action nondeterministically selects a non-zero counter and decrements it, while storing the message and the state into the local variable.

After the translation, we are left with a well-structured system. Verifying the provenance specification reduces to checking if there is a reachable configuration of the system in which `v` contains a message whose provenance automaton is in a final state. This reachability question can be solved as a *coverability problem* on the well-structured system, which is decidable. In fact, we show a symbolic encoding that gives an optimal algorithm.

3 Message Passing Programs

Preliminaries A *multiset* m over a set Σ is a function $\Sigma \to \mathbb{N}$ with finite support (i.e., $m(\sigma) \neq 0$ for finitely many $\sigma \in \Sigma$). By $\mathbb{M}[\Sigma]$ we denote the set of all multisets over Σ. As an example, we write $m = [\![\sigma_1^2, \sigma_3]\!]$ for the multiset $m \in \mathbb{M}[\{\sigma_1,\sigma_2,\sigma_3\}]$ with $m(\sigma_1) = 2, m(\sigma_2) = 0$, and $m(\sigma_3) = 1$. We write $\emptyset$ for the empty multiset, mapping each $\sigma \in \Sigma$ to 0. Two multisets are ordered by $m_1 \leq m_2$ if for all $\sigma \in \Sigma$, we have $m_1(\sigma) \leq m_2(\sigma)$. Let $m_1 \oplus m_2$ (resp. $m_1 \ominus m_2$) be the multiset that maps every element $\sigma \in \Sigma$ to $m_1(\sigma) + m_2(\sigma)$ (resp. $\max\{0, m_1(\sigma) - m_2(\sigma)\}$).

For a set X, a relation $\preceq \subseteq X \times X$ is a *well-quasi-order* (wqo) if it is reflexive, transitive, and such that for every infinite sequence $x_0, x_1, \ldots$ of elements from

X, there exists $i < j$ such that $x_i \preceq x_j$. Given a wqo $\preceq$, we define its *induced equivalence* $\equiv \subseteq X \times X$ by $x \equiv y$ if $x \preceq y$ and $y \preceq x$.

A subset X' of X is *upward closed* if for each $x \in X$, if there is a $x' \in X'$ with $x' \preceq x$ then $x \in X'$. A subset X' of X is *downward closed* if for each $x \in X$, if there is a $x' \in X'$ with $x \preceq x'$ then $x \in X'$. A function $f : X \to X$ is called $\preceq$*-monotonic* if for each $x, x' \in X$, if $x \preceq x'$ then $f(x) \preceq f(x')$.

A *transition system* $TS = (\mathcal{C}, c_0, \to)$ consists of a set $\mathcal{C}$ of configurations, an initial configuration $c_0 \in \mathcal{C}$, and a transition relation $\to \subseteq \mathcal{C} \times \mathcal{C}$. We write $\to^*$ for the reflexive transitive closure of $\to$. A configuration $c \in \mathcal{C}$ is *reachable* if $c_0 \to^* c$. A *well-structured transition system* is a $TS = (\mathcal{C}, c_0, \to)$ equipped with a well-quasi order $\preceq \subseteq \mathcal{C} \times \mathcal{C}$ such that for all $c_1, c_2, c_3 \in \mathcal{C}$ with $c_1 \preceq c_2$ and $c_1 \to c_3$, there exists $c_4 \in \mathcal{C}$ with $c_3 \preceq c_4$ and $c_2 \to c_4$.

3.1 Programming Model

Syntax We work in the setting of asynchronous message passing programs. For simplicity, we assume that the programming language has a single finitely-valued datatype $\mathcal{M}$ of messages. A *channel* is a (potentially unbounded) multiset of messages supporting two actions: a *send* action (written $ch!x$) that takes a message stored in variable x and puts it into the channel, and a *receive* action (written $ch?x$) that takes a message m from the channel and copies it to the variable x. Let C be a finite set of channels.

A *control flow graph* (CFG) $G = (X, V, E, v^0)$ consists of a set X of message variables, a set V of control locations including a unique *start location* $v^0 \in V$, and a set E of labeled directed edges between the control locations in V. Every edge in E is labeled with one of the following actions:

- an *assignment* $y := \otimes(x)$, where $x, y \in X$ and $\otimes$ is an uninterpreted unary operation on messages;
- an *assume action* $\mathsf{assume}(x = m)$, where $x \in X$ and $m \in \mathcal{M}$;
- a *send action* $ch!x$, or a *receive action* $ch?x$, where $x \in X$ and $ch \in C$.

A *message passing program* $\mathcal{P} = (Prin, C, \{G_p\}_{p \in Prin})$ consists of a finite set $Prin$ of *principals*, a set C of channels, and for each $p \in Prin$, a control flow graph G_p.

Intuitively, a message passing program consists of a finite set of processes. Each process is owned by a named entity or a principal. The processes have local variables which can be updated using unary operators, and communicate with other processes by asynchronously sending to and receiving messages from the set of channels C.

We shall use the notation $v \xrightarrow{a,p} v'$ to denote that the CFG G_p of principal p has an edge $(v, v') \in E_p$ labeled with the action a. Given the set $\{G_p\}_{p \in Prin}$ of CFGs, we define $X^\bigstar = \uplus\{X_p \mid p \in Prin\}$, $V^\bigstar = \uplus\{V_p \mid p \in Prin\}$, and $E^\bigstar = \uplus\{E_p \mid p \in Prin\}$ as the disjoint unions of local variables, control locations, and control flow edges, respectively.

Semantics We now give a *provenance-carrying* semantics to message passing programs. Let U be a (not necessarily finite) set of *provenances*. We shall associate with each message in a message passing program a provenance from U.

Let $\mathcal{P} = (Prin, C, \{G_p\}_{p \in Prin})$ be a message passing program. A *provenance domain* $\mathcal{U} = (U, \preceq, \psi)$ for $\mathcal{P}$ consists of a set U of provenances, a well-quasi ordering $\preceq$ on U, and for each principal $p \in Prin$ and for each operation $op \in \otimes \cup \{!, ?\}$, a $\preceq$-monotonic function $\psi(p, op) : U \to U$. A provenance domain is decidable if $\preceq$ is a decidable relation and ψ is a computable function. We assume all provenance domains below are decidable.

Since channels are unordered, we represent contents of a channel as a multiset of pairs of messages and provenances. A *configuration* $(\ell, \mathbf{c}, \pi)$ consists of a location function $\ell : Prin \to V^\bigstar$ mapping each principal to a control location; a channel function $\mathbf{c} : C \to \mathbb{M}[\mathcal{M} \times U]$ mapping each channel to a multiset of pairs of messages from $\mathcal{M}$ and provenances from U; and a store function $\pi : X^\bigstar \to \mathcal{M} \times U$ mapping each variable to a message and its provenance.

Define $\ell_0 : Prin \to V^\bigstar$ as the function mapping $p \in Prin$ to the start location $v_p^0 \in V_p$ and $\mathbf{c}_0 : C \to \mathbb{M}[\mathcal{M} \times U]$ as the function mapping each $ch \in C$ to the empty multiset $\emptyset$. Let $\pi_0 : X^\bigstar \to \mathcal{M} \times U$ be a mapping from variables in $X^\bigstar$ to a default initial value m_0 from $\mathcal{M}$ and a default initial provenance ε from U.

The provenance-carrying semantics of a message passing program $\mathcal{P}$ with respect to the provenance domain $(U, \preceq, \psi)$ is defined as the transition system $TS(\mathcal{P}) = (\mathcal{C}, c_0, \to)$ where $\mathcal{C}$ is the set of configurations, the initial configuration $c_0 = (\ell_0, \mathbf{c}_0, \pi_0)$, and the transition relation $\to \subseteq \mathcal{C} \times \mathcal{C}$ is defined as follows.

For a function $f : A \to B$, $a \in A$, and $b \in B$, let $f[a \mapsto b]$ denote the function that maps a to b and all $a' \neq a$ to $f(a')$. We define $(\ell, \mathbf{c}, \pi) \to (\ell', \mathbf{c}', \pi')$ if there exists $p \in Prin$ and $(\ell(p), a, \ell'(p)) \in E^\bigstar$ such that for all $p' \neq p$, we have $\ell(p') = \ell'(p')$; and

1. if $a \equiv y := \otimes(x)$ and $(m, u) = \pi(x)$ then $\mathbf{c}' = \mathbf{c}$ and $\pi' = \pi[y \mapsto (\otimes(m), \psi(p, \otimes)(u))]$;
2. if $a \equiv \mathsf{assume}(x = m)$ then $\mathbf{c}' = \mathbf{c}$, $\pi' = \pi$, and $\pi(x) = (m, \cdot)$;
3. if $a \equiv ch!x$ then $\pi' = \pi$ and if $(m, u) = \pi(x)$, then $\mathbf{c}' = \mathbf{c}[ch \mapsto \mathbf{c}(ch) \oplus [\![(m, \psi(p, !)(u))]\!]]$;
4. if $a \equiv ch?x$ and there is (m, u) such that $\mathbf{c}(ch)(m, u) > 0$ then $\mathbf{c}' = \mathbf{c}[ch \mapsto \mathbf{c}(ch) \ominus [\![(m, u)]\!]]$ and $\pi' = \pi[x \mapsto (m, \psi(p, ?)(u))]$.

Intuitively, in each step, one of the principals executes a local action. An assignment action $y := \otimes(x)$ transforms the message contained in x by applying the operation $\otimes$ and transforms the provenance of x by applying ψ, storing the new message and its provenance in y. An assume checks that a variable has a specific message. Sends and receives model asynchronous communication to shared channels. Send actions are non-blocking, receive actions are blocking, and a channel can reorder messages.

Let $\mathcal{P}$ be a message passing program and $\mathcal{U} = (U, \preceq, \psi)$ a provenance domain. We consider provenance specifications given by downward closed sets over U. Downward closed sets capture the "monotonicity" property that holds in many domains. For example, a security policy that holds when a given set of trusted principals looks at a message, is also met when fewer principals look at it. Conversely, bad behaviors are captured by upward closed sets.

The *provenance verification problem* asks, given a variable x of $\mathcal{P}$ and a downward closed set $D \subseteq U$, if the provenance of the content of variable x is always in D along all runs of the program. Dually, the specification is violated if there exists a reachable configuration where the provenance of variable x is in the upward closed set $I = U \backslash D$. Such a configuration indicates a violation of security policies. We shall use the dual formulation in our algorithms.

3.2 Examples

We now give illustrative examples of provenance domains.

Example 1. **[The Language Provenance Domain]** Consider $U = Prin^*$, the set of finite sequences over principals. Let $(Q, Prin, q_0, \delta)$ be a deterministic finite automaton, and let $\preceq$ be defined as $u \preceq v$ iff $\delta(q_0, u) = \delta(q_0, v)$. Let ψ be the function defined as $\psi(p, !)(u) = u \cdot p$, and $\psi(\cdot, \cdot)(u) = u$ for all other operations. Intuitively, the language provenance domain associates a list of principals with each message: the sequence of principals who have sent this message along the current computation.

A downward closed set D in the language provenance domain is a regular language that prescribes a set $F \subseteq Q$ of final states for the finite automaton A. The corresponding upward closed set I is a regular language that prescribes a set $Q \setminus F$ of final states for the complement automaton $\overline{A}$. The provenance verification problem asks, for example, if the provenance of the message in p3 always belongs to the regular language Patient Secretary Doctor$^+$ along all runs of the program.

Example 2. **[The Set Provenance Domain]** Let $U = 2^{Prin}$, the set of sets of principals. Let $\preceq$ be set inclusion. Since the set of principals is finite, this is a wqo. Let ψ be the function defined as $\psi(p, !)(u) = u \cup \{p\}$, and $\psi(\cdot, \cdot)(u) = u$ for all other operations. The set provenance domain associates a set of principals with each message: the set contains all the principals who have sent this message (potentially multiple times). An upward closed set I corresponds to a set of sets of principals, such that if a set of principals is in I, each of its supersets is also in I. As an example, suppose the set of principals *Prin* is divided into "trusted" and "untrusted" principals. A downward closed set D specifies the sets all of whose elements are "trusted". As a result, the corresponding upward closed set I captures all sets containing at least one "untrusted" principal. The provenance verification problem asks, given a variable x, if there is a message stored in x along a run that has a provenance which is one of the sets in I.

4 Model Checking

We now give a model checking algorithm for provenance verification by reduction to labeled Petri nets.

4.1 Labeled Petri Nets

A *Petri net* (PN) is a tuple $N = \langle S, T, (I, O) \rangle$ where S is a finite set of places, T is a finite set of transitions, and functions $I : T \to S \to \{0, 1\}$ and $O : T \to S \to \{0, 1\}$ encodes pre- and post-conditions of transitions.

A *marking* is a multiset over S. A transition $t \in T$ is *enabled* at a marking μ, denoted by $\mu[t\rangle$, if $\mu \geq I(t)$. An enabled transition t at μ may *fire* to produce a new marking μ', denoted by $\mu[t\rangle\mu'$, where $\mu' = \mu \ominus I(t) \oplus O(t)$. We naturally lift the enabledness and firing notions from one transition to a sequence $\sigma \in T^*$ of transitions. A PN N and a marking μ_0 define a transition system $TS(N) = (\mathbb{M}[S], \mu_0, \rightarrow)$, where $\mu \rightarrow \mu'$ if there is a transition t such that $\mu[t\rangle\mu'$.

The encoding of a PN N is given by a list of pairs of lists. Each transition $t \in T$ is encoded by two lists corresponding to $I(t)$ and $O(t)$. Each list $I(t)$ or $O(t)$ is encoded as a bitvector of size $|S|$. The size of N, written $\|N\|$, is the sum of the representations of all the lists.

Let N be a Petri net and μ_0 and μ markings. The *coverability problem* asks if there is $\mu' \geq \mu$ that is reachable from μ_0, so $\mu_0 \rightarrow^* \mu' \geq \mu$. In this case, we say μ is coverable from μ_0.

Theorem 1. *[18,24] The coverability problem for Petri nets is EXPSPACE-complete.*

In the usual definition of Petri nets, tokens are simply uninterpreted "dots" and markings count the number of dots in each place. We now extend the Petri net model with tokens labeled with elements from a decidable provenance domain $\mathcal{U}$. A $\mathcal{U}$-labeled Petri net $N = \langle S, T, (I, O), \Lambda\rangle$ is a Petri net $\langle S, T, (I, O)\rangle$ that is equipped with a labeling function Λ specifying how provenance markings are updated when a transition is fired. Consider a transition $t \in T$. Let $p_1, \ldots, p_k$ be an ordering of all the places in S for which $I(t)(p) = 1$. For each place $p' \in S$ with $O(t)(p') = 1$, the labeling function $\Lambda(t, p')$ is a $\preceq$-monotonic function $U^k \rightarrow U$. We assume the labeling function Λ is computable.

A *labeled marking* μ is a mapping from places S to multisets over U, i.e., it labels each token in a marking with an element of U. A labeled marking μ induces a marking $\mathsf{erase}(\mu)$ that maps each $p \in S$ to $\sum_{u \in U} \mu(p)(u)$ obtained by erasing all provenance information carried by tokens. Fix a transition t, and let $p_1, \ldots, p_k$ be an ordering of the places such that $I(t)(p) = 1$. The transition t is enabled at a labeled marking μ if for each $p \in S$ with $I(t)(p) = 1$, we have $\mathsf{erase}(\mu)(p) \geq 1$. An enabled transition t at μ can fire to produce a new labeled marking μ', denoted (by abuse of notation) $\mu[t\rangle\mu'$, defined as follows. To compute μ' from μ, first pick and remove arbitrarily tokens from p_1 to p_k with labels u_1 to u_k respectively. Then, for each p' with $O(t)(p') = 1$, add a token whose label is $\Lambda(t, p')(u_1, \ldots, u_k)$ to p'. All other places remain unchanged. We extend the firing notion to sequences of transitions, as well as notions of transition system, size, reachability, and coverability to labeled Petri nets in the obvious way.

To prove the coverability problem is decidable for $\mathcal{U}$-labeled Petri nets, we argue that their transition systems $(\mathbb{M}[U]^S, \mu_0, \hookrightarrow)$ are *well-structured* in that the labeled markings can be equipped with an order that allows larger labeled markings to mimic the behaviour of smaller ones, i.e. there is a wqo $\ll \subseteq \mathbb{M}[U]^S \times \mathbb{M}[U]^S$ that is compatible with the transitions: for all $\mu_1 \hookrightarrow \mu_1'$ and $\mu_1 \ll \mu_2$ there is $\mu_2 \hookrightarrow \mu_2'$ so that $\mu_1' \ll \mu_2'$.

To define a suitable wqo on labeled markings, we first compare the multisets on a place. Intuitively, $\mu(p) \ll \mu'(p)$ with $\mu, \mu' \in \mathbb{M}[U]^S$ and $p \in S$ if for every

u in $\mu(p)$ there is an element u' in $\mu'(p)$ such that $u \preceq u'$ in the wqo $\preceq$ of the provenance domain. Hence, $\mu \ll \mu'$ if for each $p \in S$ there is an injective function $f_p : \mu(p) \to \mu'(p)$ so that for each $u \in \mu(p)$, we have $u \preceq f_p(u)$. The result is a wqo by Higman's lemma [13] and the fact that wqos are stable under Cartesian products. The ordering is also compatible with the transitions by the monotonicity requirement on labelings. The following theorem follows using standard results on well-structured transition systems [1,8].

Theorem 2. *The coverability problem for $\mathcal{U}$-labeled Petri nets is decidable and EXPSPACE-hard for decidable provenance domains $\mathcal{U}$.*

The coverability problem for labeled Petri nets need not be in EXPSPACE, even when the operations on $\mathcal{U}$ are provided by an oracle. For example, nested Petri nets [20] can encode reset nets, for which a non-primitive recursive lower bound is known for coverability [26].

4.2 From Message Passing Programs to Labeled Petri Nets

Let $\mathcal{P} = (Prin, C, \{G_p\}_{p \in Prin})$ be a message passing program and $\mathcal{U} = (U, \preceq, \psi)$ a provenance domain. We now give a labeled Petri net semantics to the program.

Define the labeled Petri net $N(\mathcal{P},\mathcal{U}) = \langle S, T, (I, O), \Lambda\rangle$ as follows. There is a place for each program location, for each local variable and message value, and each channel and message value: $S = V^\bigstar \cup (X^\bigstar \times \mathcal{M}) \cup (C \times \mathcal{M})$.

In the definition of labels, we use variable $\mathsf{prov}(p)$ for the token (which is a provenance) in place $p \in S$ that is used for firing. The set T is the smallest set that satisfies the following conditions.

1. For each $e \equiv v \xrightarrow{y:=\otimes(x),p} v'$ in $E^\bigstar$, and for each $m, m' \in \mathcal{M}$, there is a transition t with $I(t)=[\![v,(x,m),(y,m')]\!]$ and $O(t)=[\![v',(x,m),(y,\otimes m)]\!]$. Also, $\Lambda(t,(x,m))=\mathsf{prov}(x,m)$, $\Lambda(t,(y,\otimes m))=\psi(p,\otimes)(\mathsf{prov}(x,m))$, and $\Lambda(t,v')=\varepsilon$.
2. For each $e \equiv v \xrightarrow{\mathsf{assume}(x=m),p} v'$ in $E^\bigstar$, there is a transition t with $I(t)=[\![v,(x,m)]\!]$ and $O(t)=[\![v',(x,m)]\!]$. Also, $\Lambda(t,v') = \varepsilon$, and $\Lambda(t,(x,m)) = \mathsf{prov}(x,m)$.
3. For each $e \equiv v \xrightarrow{ch!x,p} v'$ in $E^\bigstar$, and for each $m \in \mathcal{M}$, there is a transition t with $I(t)=[\![v,(x,m)]\!]$, $O(t)=[\![v',(x,m),(ch,m)]\!]$. Also, $\Lambda(t,v')=\varepsilon$, $\Lambda(t,(x,m))=\mathsf{prov}(x,m)$, and $\Lambda(t,(ch,m))=\psi(p,!)(\mathsf{prov}(x,m))$.
4. For each $e \equiv v \xrightarrow{ch?x,p} v'$ in $E^\bigstar$, for each $m, m' \in \mathcal{M}$, there is a transition t with $I(t) = [\![v,(x,m),(ch,m')]\!]$ and $O(t) = [\![v',(x,m')]\!]$. Also, $\Lambda(t,v') = \varepsilon$ and $\Lambda(t,(x,m')) = \psi(p,?)(\mathsf{prov}(ch,m'))$.

To relate $\mathcal{P}$ with its Petri nets semantics $N(\mathcal{P},\mathcal{U})$, we define a bijection ι between configurations and labeled markings: $\iota(\ell,\mathbf{c},\pi) = \mu$ iff all of the three conditions hold: (1) $\mu(v) = [\![\varepsilon]\!]$ iff there is $p \in Prin$ with $\ell(p) = v$; (2) for all $x \in X^\bigstar$, for all $m \in \mathcal{M}$, and for all $u \in U$, $\mu(x,m) = [\![u]\!]$ iff $\pi(x) = (m,u)$; (3) for all $ch \in C$, for all $m \in \mathcal{M}$, and for all $u \in U$, $\mu(ch,m)(u) = k$ iff $\mathbf{c}(ch)(m,u) = k$. Define the initial labeled marking $\mu_0 = \iota(\ell_0,\mathbf{c}_0,\pi_0)$. The following observation follows from the definition of ι.

Lemma 1. *$TS(\mathcal{P})$ and $TS(N(\mathcal{P},\mathcal{U}))$ are isomorphic.*

Complexity-wise, the problem inherits the hardness of coverability in (unlabeled) Petri nets for any non-trivial provenance domain.

Theorem 3. *Given a message passing program $\mathcal{P}$ and a decidable provenance domain $\mathcal{U} = (U, \preceq, \psi)$, the provenance verification problem is decidable. It is EXPSPACE-hard for any provenance domain with at least two elements.*

Proof. From the construction of the labeled Petri net, Lemma 1, the provenance verification problem is reducible in polynomial time to coverability for labeled Petri nets. Thus, by Theorem 2, provenance verification problem is decidable.

For EXPSPACE-hardness, we reduce Petri net coverability to provenance verification. To simulate a Petri net with a message passing program, we introduce a channel for every place and then serialize the reading of tokens. Consider $N = \langle S, T, (I, O)\rangle$. We construct a message passing program with one principal, one message, and a channel for each place in S. The control flow graph of the only principal has a central node from which loops simulate the Petri net transitions. At each step, the central node picks a transition $t \in T$ non-deterministically and simulates first the consumption and then the production of tokens — one by one. To consume a token from place p with $I(t)(p) = 1$, the principal receives a message from channel p. For the production, it sends a message to the channel p' with $O(t)(p') = 1$. Additionally, the principal non-deterministically checks if the current configuration of channels covers the target marking. If so, it writes a message into a special variable x. The provenance verification problem asks whether x ever contains a message with non-trivial provenance. EXPSPACE-hardness follows from Theorem 1. ∎

4.3 EXPSPACE Upper Bounds

For set and language provenance domains, we can in fact show a matching upper bound on the complexity. It relies on a fairly general product construction and reduction to Petri nets. We say that a provenance domain $\mathcal{U}$ is of *finite index* if the equivalence induced by $\preceq$ has finitely many classes. We denote this equivalence by $\equiv$. Clearly, any finite provenance domain (thus, the set domain) is of finite index. The language domain is also of finite index: take the equivalence relation induced by the Myhill-Nerode classes of the language. The following lemma characterizes the structural properties of provenance domains of finite index.

Lemma 2. *Consider a Petri net $N = \langle S, T, (I, O), \Lambda\rangle$ that is labelled by $\mathcal{U}$ of finite index. (1) The equivalence classes are closed under Λ: for any tuple $e_1, \ldots, e_k$ of $\equiv$-equivalence classes, the image $\Lambda(e_1, \ldots, e_k)$ is fully contained in another equivalence class e. (2) The upward-closure of any $u \in U$ is a finite union of $\equiv$-classes.*

Let $N = \langle S, T, (I, O), \Lambda\rangle$ be a $\mathcal{U}$-labeled Petri net, and suppose $\mathcal{U}$ is of finite index. We now define a product construction that reduces N to an ordinary Petri net $N' = \langle S', T', (I', O')\rangle$. Intuitively, for each place $p \in S$ and each

equivalence class e, there is a place (p, e) in S' that keeps track of all tokens in N at place p and having their label in the equivalence class e. We define $S' = S \times \{[u]_{\equiv} \mid u \in U\}$. Each transition in N is simulated by a family of transitions in T', one for each combination of equivalence classes for the source tokens. More precisely, T' is the smallest set that contains the following family of transitions for each $t \in T$. Let $p_1, \ldots, p_k$ be the places in S with $I(t)(p_i) = 1$. For each sequence $\overline{p} = \langle e_1, \ldots, e_k \rangle$ of k-tuples of $\equiv$-equivalence classes, we have a transition $t_{\overline{p}} \in T'$ such that $I'(t_{\overline{p}})((p_i, e_i)) = 1$ for $i = 1, \ldots, k$ and $I'(t_{\overline{p}})(p) = 0$ for all other places. Moreover, for each $p \in S$ with $O(t)(p) = 1$ labeled with Λ, we have that $O'(t_{\overline{p}})((p, e)) = 1$ with $\Lambda(e_1, \ldots, e_k) \subseteq e$. Note that this inclusion is well-defined by Lemma 2(1). This product construction reduces a labelled coverability query in N to several unlabelled queries in N'. What are the unlabelled queries we need? Consider a token u in a labelled marking $\mu \in \mathbb{M}[U]^S$. We use the equivalence classes that, with Lemma 2(2), characterize the upward closure of u. In the following proposition, we assume that these classes are effectively computable. This is the case for set and language domains.

Proposition 1. *If $\mathcal{U}$ is of finite index, coverability for $\mathcal{U}$-labeled Petri nets is reducible to coverability for Petri nets.*

Proposition 1 provides a 2EXPSPACE upper bound for the set and language domains, which is not optimal. Consider the set domain. Each subset of principals yields an equivalence class of provenances. Hence, there is an exponential number of classes and the above product net is exponential. A similar problem occurs for the language domain if the provenance specification is given by a non-deterministic finite automaton. There are regular languages where this non-deterministic representation is exponentially more succinct than any deterministic one. The deterministic one, however, is needed in the product. To derive an optimal upper bound, we give compact representations of these exponentially many classes.

Theorem 4. *Provenance verification problem is in EXPSPACE for set and language domains.*

Proof. To establish membership in EXPSPACE, we implement the above reduction from labeled to unlabeled coverability in a compact way, so that the size of the resulting Petri net is polynomial in the size of the input. The challenge is to avoid the multiplication between places and equivalence classes, which may be exponential. Instead, we first encode the classes into polynomially many additional places, and maintain the relationship between a place and a class in the marking of the new net. Second, we only keep the provenance information for tokens in the goal marking, and omit the provenance of the remaining tokens.

Let E be the set of equivalence classes of a provenance domain of finite index. Let $\kappa = \lceil \log |E| \rceil$. The symbolic representation of E uses 2κ places. Let the places be $b_0, d_0, \ldots, b_{\kappa-1}, d_{\kappa-1}$. We maintain the invariant that in any reachable marking, exactly one of b_i, d_i contains a single token, for $i = 0, \ldots, (\kappa - 1)$. Intuitively, a token in b_i specifies the bit i is one, and a token in d_i specifies

the bit i is zero. Using constructions on (1-safe) Petri nets, one can "copy" a bitvector, remove all tokens from a bitvector, or update a bitvector to a value.

For example, to empty out a bitvector, we introduce $\kappa + 1$ places $p_0, \ldots, p_\kappa$, with an initial token in p_0. Each p_i, $i \in \{0, \ldots, \kappa - 1\}$, has two transitions: they take a token from p_i and from b_i (resp. d_i), and put a token in p_{i+1}. When p_κ is marked, all the bits have been cleared. Similarly, to copy the configuration from places $b_0, d_0, \ldots, b_{\kappa-1}, d_{\kappa-1}$ to empty places $b'_0, d'_0, \ldots, b'_{\kappa-1}, d'_{\kappa-1}$, we use the following gadget. We add additional $\kappa+1$ places $p_0, \ldots, p_\kappa$, with an initial token on p_0. For each p_i, $i \in \{0, \ldots, \kappa - 1\}$ there are two transitions: one takes a token from p_i and one token from b_i and puts a token in p_{i+1}, one in b_i, and one in b'_i; the other takes a token from p_i and one from d_i and puts a token in p_{i+1}, one in d_i, and one in d'_i. When the place p_κ is marked, the bits in $b_0, d_0, \ldots, b_{\kappa-1}, d_{\kappa-1}$ have been copied to $b'_0, d'_0, \ldots, b'_{\kappa-1}, d'_{\kappa-1}$.

Now, in the translation of the Petri net, instead of a place (x, m, e) for each variable x, message m, and equivalence class $e \in E$, we keep 2κ places for each place (x, m), encoding the equivalence class e for x and m. If all 2κ places for (x, m) are empty in a marking, it implies that the current content of x is not m; otherwise, the provenance equivalence class $e \in E$ of (x, m) is encoded by the 2κ bits. The transitions of the net are updated with the gadgets to copy the provenance bitvectors in case of assignments.

Moreover, for each channel ch, we maintain the provenance information of one message, and drop the provenance of every other message in the channel. That is, each channel ch is modeled using places (ch, m) for each $m \in \mathcal{M}$, and in addition, $2\kappa \cdot |\mathcal{M}|$ places that encode the provenance equivalence class of one message for each value in $\mathcal{M}$ stored in the channel. Intuitively, tokens in (ch, m) denote messages with value m in the channel ch whose provenance has been "forgotten" and tokens in the bitvectors encode one message (per message type) in the channel whose provenance is encoded using 2κ places. We use non-determinism to guess which messages contribute to the message with provenance in the target. When a message is sent to a channel, we non-deterministically decide to keep its provenance (thus using the bitvectors, moving any tokens already there) or to drop its provenance.

Similarly, when we receive from a channel, we non-deterministically decide to either read from the "special" places for the encoding of an equivalence class, or from the "normal" place.

Now, for the set domain, we use $2|\mathit{Prin}|$ places to encode sets of principals. For the language domain, where the specification is given by a non-deterministic automaton with states Q, we use $2|Q|$ places to encode the subsets of states. The encoding allows us to perform the subset construction on the fly. Each action of the program requires at most a polynomial number of additional places to encode the gadgets. Thus, we get a Petri net that is polynomial in the size of the message passing program and the specification. Thus, using Theorem 1, we get the EXPSPACE upper bound. ∎

5 Implementation and Experiments

We have implemented a tool for the provenance verification problem for language provenance domains. Our tool takes as input a message passing program encoded in an extended Promela syntax in which channels are marked asynchronous and have the semantics described in Section 3. It reduces the provenance verification problem to Petri net coverability using the algorithm from Section 4. We first used state-of-the-art tools for Petri net coverability [9,21]. Unfortunately, the times taken to verify the provenance properties were high. This is because Petri net coverability tools are optimized for nets with many places that can be unbounded and for high concurrency. Instead, message passing programs only have few places that are unbounded (the channels). Our second observation is that message passing programs have a lot of scope for partial-order reduction, by allowing a process to continue executing until it hits a blocking receive action. To take advantage of these features, we implemented a coverability checker that combines expand-enlarge-check (EEC) [10] with partial order reduction [11].

5.1 Expand-Enlarge-Check and Partial Order Reduction

The EEC procedure [10] performs *counter abstraction* over a Petri net. We observe that only the places representing shared channels can have more than one token in our Petri nets. Instead of counting the exact number of messages in a channel, we fix a parameter $k \geq 0$ and count precisely up to k. If at any point, the number of messages in a channel exceeds k, we replace the number by ∞. Once the count goes to ∞, we do not decrease the count even when messages are removed from the channel. For example, if $k = 0$, the abstraction of a channel distinguishes two cases: either the channel has no messages or it has an arbitrary number of messages.

The abstraction is sound, in that if a marking is coverable in the original net, it is also covered in the abstraction. However, the abstraction can add spurious counterexamples, in that a marking can be considered coverable in the abstraction, even though it is not coverable in the original net. By concretely simulating a specific counterexample path, we can decide if the counterexample is genuine or spurious. In case the counterexample is spurious, we increase the parameter k and continue. This abstraction-refinement process is guaranteed to terminate, by either finding a genuine path that covers a given marking, or by proving that the target marking is not coverable for some parameter k in the abstraction [10]. We have found that $k = 1$ is usually sufficient to soundly abstract the state space and to prove a provenance property; this is consistent with other uses of counter abstractions in verification [23,17].

Additionally, we note that once the parameter k is fixed, the state space of the system is finite, since each channel can have at most $k + 2$ messages ($\{0, \ldots, k\} \cup \{\infty\}$). Thus, for each k, we can perform reachability analysis using a finite-state reachability engine. In our implementation, we choose the Spin model checker [14] to perform reachability analysis in every iteration where k is fixed. In Spin models, for each channel, each message type, and each state of the

provenance automaton, we have a variable that takes $k+2$ values, implementing the k-abstraction.

Additionally, message passing programs have the potential for partial order reduction. For example, each process in the program can be executed until it reaches a blocking receive action, and the local actions of different processes commute. Since Spin already implements partial order reduction, we get the benefits of partial order reduction for free.

5.2 Case Studies: Message Passing Benchmarks

We first describe our evaluation on a set of three message passing systems (see Table 1). The example MyHealth Portal is described in [2]. We checked if the provenance of a variable is always in the regular language $\mathsf{Patient}\ (\mathsf{Secretary} + \varepsilon)\ \mathsf{Nurse}\ \mathsf{Doctor}^{+} + \varepsilon$. The bug tracking system [16] manages software bug reports. It has five principals and eight types of messages (bug report, closed, fix-again, fix, must-fix, more-information, pending, and verified). The provenance specification, given as an automaton with nine states, encodes the flow of events leading from a bug report to a bug fix. We found that the original system violated the specification because a message was sent to an incorrect channel. After fixing the bug, we were able to prove the property for the new system. The Service Incident Exchange Standard (SIS) specifies a system to share service incident data and facilitate resolutions. The standard envisages interactions between service requesters and providers. We took the system model from [4], which consists of 16 principals, 18 channels, and 9 message types. The property to check is once a service request is terminated, it is never reopened.

Results Table 2 lists the analysis results. All experiments were performed on a 2 core Intel Xeon X5650 CPU machine with 64GB memory and 64bit Linux (Debian/Lenny). We compare state-of-the-art Petri net coverability tools (Mist2 [9] and Petruchio [21]) with our Spin-based coverability checker. We run Petruchio and three different options of Mist2 and report the best times. A timeout indicates that all the tools timed out. The "Markings" row indicates the number of coverability checks required to prove correctness. The time denotes the sum of the times for all the coverability checks to finish, where for each check, we take the best time by any tool.

For our Spin-based checker, we report the parameter k for which either a genuine counterexample was found, or the system was proved correct. We compare the results with and without partial order reduction. For each run, we give three numbers: the number of states and transitions explored by our checker and the time taken. There is a significant reduction when partial order reduction is turned on. Moreover, our Spin-based implementation is orders of magnitude faster than the Petri net coverability tools.

5.3 Private Mode and Firefox Extensions

We performed a larger case study on provenance in browser extensions. Modern browsers provide a "private mode" that deletes cookies, forms, and browsing history at the end of each browsing session. Browsers also provide an extension mechanism, through which third-party developers can add functionality to

Example	Principals	Messages	Channels	Automaton
Health Care	4	4	5	6
Bug Tracking	5	8	5	9
SIS	16	9	18	2

Table 1. Message passing benchmarks. "Principals" is the number of principals, "Messages" the possible values of messages, "Channels" is the number of shared channels, and "Automaton" is the number of states in the provenance automaton.

PN tools	Health Care	Bug Tracking (1)	Bug Tracking (2)	SIS
Markings	12	1	40	127
Time	125.6s	2308.940s	timeout	1152.07s
Our Checker	**Health Care**	**Bug Tracking (1)**	**Bug Tracking (2)**	**SIS**
k	0	1	0	1
States (No POR)	6351	39	4905516	3738754
States (POR)	2490	39	995468	893786
Trans (No POR)	23357	39	24850365	17274836
Trans (POR)	4249	39	1707682	1736062
Time (No POR)	0.04s	0.01s	38.6s	58.7s
Time (POR)	0.01s	0.01s	3.37s	6.10s

Table 2. Results of the message passing benchmarks. Bug Tracking (1) is the buggy version.

browsers. Extensions can communicate between their front- and back-ends by asynchronous messages passing, and between each other via temporary files. Moreover, Firefox lets extension developers manage SQLite databases in user machines by invoking a service called *mozIStorageService*. It provides a set of asynchronous APIs for extensions to communicate with databases through SQL queries. If extension developers do not properly handle the private mode, user data may be stored in the database while the user is browsing in private mode.

It is expected that browser extensions should respect the private mode. Unfortunately, browsers do not restrict an extension's capability in private mode, and it is the responsibility of developers not to record user data in private mode. In the second set of case studies, we check if extension developers for Firefox obey the privacy concerns when the user is browsing in private mode.

Our goal is to check if extensions using *mozIStorageService* can store user data while in private mode. We formulate the problem of tracking information flow in private mode as a provenance verification problem. Consider a set of browser extensions cooperating with each other, and a principal Db modelling a database. For each extension A, we introduce two principals NormA and PrivA that represent two instances of A running in the normal and in the private mode, respectively. For each extension A that saves data to the database, there are two channels $ch_{\mathsf{Db}}, ch'_{\mathsf{Db}}$ for NormA and PrivA to interact with Db. Moreover, for each pair of extensions (A, B) where A sends data to B, for instance, by writing and reading files, there are four combinations: $(\mathsf{NormA}, \mathsf{NormB})$, $(\mathsf{PrivA}, \mathsf{NormB})$, $(\mathsf{NormA}, \mathsf{PrivB})$, and $(\mathsf{PrivA}, \mathsf{PrivB})$. For each case, we introduce a channel ch to model the message flow from A to B. The prop-

erty we check is whether some PrivA directly or indirectly updates the database. Note that it is not sufficient to ensure every write to the database is guarded by a check that the browser is not in private mode. There can be indirect flows where data is stored in a temporary file in private mode, or communicated to a different extension, and later stored in the database.

We use Firefox 13.0.1 in our experiments. We selected nine popular extensions from Firefox's extension repository, by filtering them based on the keywords *form*, *history*, and *shopping*, and then filtering based on their use of *mozIStorageService*. The extensions we chose have about 50000 users on average.

Our tool works as follows. We first use JSure [7], a Javascript parser and static analyzer, to obtain the control flow from the extension source code, and to produce a message passing program in Promela syntax. As the access to a database is either via calling the *mozIStorageService* APIs directly or via helper extensions, we capture along the control flow the information about when an extension calls these APIs to update the database, and the information about when extensions communicate with each other by writing and reading temporary files. Our front end abstracts away complex data structures in the program. In particular, we do not track the contents inserted into the database. This may lead to false positives in the analysis. We then run our Spin-based back-end to verify the message passing program.

Table 3 lists the results. Five out of the nine examples are found to store user information even in private mode. All examples can be verified efficiently (in a few milliseconds) because usually a small portion of code is related to database accesses and extension communications, and complex data structures are abstracted out. For all unsafe cases, we have successfully replayed executions that violate the private mode in Firefox.

6 Extensions

We have described a general algebraic model of provenance in concurrent message passing systems and an algorithm for statically verifying provenance properties. For these expressive programs, only dynamic checks or imprecise static checks had been studied so far. While the complexity may seem high, reachability analysis in message passing programs is already EXPSPACE-complete, so provenance verification does not incur an extra cost.

Our decidability results continue to hold under some extensions to the programming model. For example, our decidability results also hold when programs can test the provenance of a message against an upward closed set in a conditional, or in the presence of a spawn instruction that dynamically generates a new thread of execution. Informally, to decide provenance verification in the presence of provenance-tests, we extend the product construction to track the membership in each upward closed set appearing syntactically in some conditional. To handle spawn, we modify the reduction to Petri nets to keep a place for each spawned instance (that is, each tuple of control location and valuation to local variables).

On the other hand, many other extensions are easily seen to be undecidable. For example, if each principal executes a recursive program, or if messages come

Name	LOC	Leak	Usage	Leak Details	Time
Amazon Price History and More 4.1.4	8124	Yes	Provide comparative pricing for searched products. Inform pricing drops for searched products.	Records shopping history while in private mode.	57ms
Facebook Chat History Manager 1.5	2798	Yes	Help users organize conversations by time and names of persons.	Records the person to whom users talk, the conversation content, and the time in private mode.	60ms
FVD Speed Dial with Online Sync 4.0.3	21278	Yes	Provide a dashboard holding favorite websites of users. Cross-platform bookmark synchronization.	Keeps counting how often users look at the websites on their Speed Dial in private mode and lists them.	57ms
Privad 1.0	17593	Yes	Uses differential privacy to prevent ad targeting.	Records user browsing history while in private mode.	60ms
Shopping Assist 3.2.4.6	15263	Yes	Provide comparative pricing for searched products.	Records shopping history while in private mode.	57ms
Form History Control 1.2.10.3	16560	No	Autosave text on forms, search bar history, for crash recovery.		63ms
History Deleter 2.4	3027	No	Utilities to delete history automatically by user defined rules.		90ms
Lazarus: Form Recovery 2.3	10839	No	Autosave text on forms, search bar history, for crash recovery.		64ms
Session Manager 0.7.9	14010	No	Autosave sessions by time for crash recovery.		104ms

Table 3. Experimental results for Firefox extensions.

from an unbounded domain such as the natural numbers, or if channels preserve the order of messages, the provenance verification problem becomes undecidable by simple reductions from known undecidable problems [22].

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS '96*, pages 313–321. IEEE, 1996.
2. A. Barth, J. Mitchell, A. Datta, and S. Sundaram. Privacy and utility in business processes. In *CSF*, pages 279–294. IEEE, 2007.
3. P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *ICDT*, LNCS 1973, pages 316–330. Springer, 2001.
4. S. Chaki, S. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *POPL*, pages 45–57. ACM, 2002.

5. J. Cheney, A. Ahmed, and U. Acar. Provenance as dependency analysis. *Math. Struct. in Computer Science*, 21:1301–1337, 2011.
6. Y. Cui, J. Widom, and J. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25:179–227, 2000.
7. B. Durak. JSure. Available at https://github.com/berke/jsure.
8. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.
9. P. Ganty, J.-F. Raskin, and L. V. Begin. From many places to few: Automatic abstraction refinement for Petri nets. *Fund. Informaticae*, 88(3):275–305, 2008.
10. G. Geeraerts, J.-F. Raskin, and L. Van Begin. Expand, enlarge and check: new algorithms for the coverability problem of WSTS. In *FSTTCS '04*, LNCS 3328, pages 287–298. Springer, 2004.
11. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. LNCS 1032. Springer, 1996.
12. T. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40. ACM, 2007.
13. G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.* (3), 2:326–336, 1952.
14. G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
15. 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 *WWW*, pages 40–52, 2004.
16. J. Janák. Issue tracking systems. Diplomová práce, Masarykova univerzita, Fakulta informatiky, 2009.
17. R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL '07*, pages 339–350. ACM, 2007.
18. R. Lipton. The reachability problem is exponential-space hard. Technical Report 62, Department of Computer Science, Yale University, 1976.
19. B. Livshits and M. Lam. Finding security errors in Java programs with static analysis. In *Usenix Security Symposium*, pages 271–286, 2005.
20. I. Lomazova and P. Schnoebelen. Some decidability results for nested Petri nets. In *Ershov Memorial Conference*, LNCS 1755, pages 208–220. Springer, 2000.
21. R. Meyer and T. Strazny. Petruchio: From dynamic networks to nets. In *CAV*, LNCS 6174, pages 175–179. Springer, 2010.
22. M. Minsky. *Finite and Infinite Machines*. Prentice-Hall, 1967.
23. A. Pnueli, J. Xu, and L. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In *CAV*, LNCS 2404, pages 107–122. Springer, 2002.
24. C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.
25. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21:5–19, 2003.
26. P. Schnoebelen. Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In *MFCS*, LNCS 6281, pages 616–628. Springer, 2010.
27. I. Souilah, A. Francalanza, and V. Sassone. A formal model of provenance in distributed systems. In *Workshop on the Theory and Practice of Provenance*, 2009.