PolSim: Automatic Policy Validation via Meta-Data Flow Simulation

Mohamed Alzayat[1,2,3]
alzayat@mpi-sws.org

Deepak Garg[1,3]
dg@mpi-sws.org

Peter Druschel[1,3]
druschel@mpi-sws.org

[1]Max Planck Institute for Software Systems
[2]Max-Planck-Institut für Informatik (IMPRS-CS)
[3]Universität des Saarlandes
Saarland Informatics Campus (SIC)
Saarbrücken, Germany

ABSTRACT

Every year millions of confidential data records are leaked accidentally due to bugs, misconfiguration, or operator error. These incidents are common in large, complex, and fast evolving data processing systems. Ensuring compliance with data policies is a major challenge. Thoth [17] is an information flow control system that uses coarse-grained taint tracking to control the flow of data. This is achieved by enforcing relevant declarative policies at processes boundaries. This enforcement is applicable regardless of bugs, misconfiguration, and compromises in application code, or actions by unprivileged operators. Designing policies that make sure all and only compliant flows are allowed remains a complex and error-prone process. In this paper, we introduce PolSim, a simulation tool that aids system policy designers by validating the provided policies and systematically ensuring that the system allows all and only expected flows. Our proposed simulator approximates the dynamic run-time environment, semi-automatically suggests internal flow policies based on data flow, and provides debugging hints to help policy designers develop a working policy for the intended system before deployment.

1. INTRODUCTION

Data retrieval systems such as search engines, social networks, online data sharing systems, governmental, and healthcare data systems have sensitive data that is subject to different policies. Enforcing compliance with data policies requires well engineered systems that guarantee confidential data is not leaked due to a bug, misconfiguration, or compromise in a large and evolving application code-base.

Online search engines such as Google Search or Microsoft Bing, for instance, index a tremendous amount of online content, each subject to its own usage policy. Examples of indexed content are many. For example, emails are subject to a policy restricting access to only owners, corporate documents are only readable by employees, and unpublished research papers are only accessible to their authors. Search engines have policies regarding the anonymity of queries and the privacy of personalization profiles. Moreover, search results must comply with censorship requirements in some countries.

Similarly, social networks such as Facebook or Google+ have privacy policies for relevant content, such as user posts, likes, and photos. Such content can be, for instance, deemed private, limited to friends, or friends of friends, or made public. Other services, such as private messaging have privacy policies giving access only to the parties engaging in a conversation.

Ensuring that such policies are properly enforced is a labor-intensive and error-prone process [32]. Enforcing these policies requires modifications across multiple layers in the data retrieval system, making it hard to reason about and prove the correctness of the enforcement mechanisms. Reports of data breaches are increasing [7, 6, 1, 3, 8, 10, 9, 34]. These data breaches cause customer disappointment, company embarrassment, loss of reputation and sometimes lead to fines. This affects business giants as well as smaller companies. When it comes to governments, data breaches are even more serious as the leaked data puts people at the risk of identity theft as reported by the Philippines commission on elections [8]. Therefore, developing mechanisms to enforce policies in data retrieval systems is crucial. Researchers at Microsoft Research and Carnegie Mellon University have developed the Grok system, which combines lightweight static analysis with the semi-automatic annotation of source code to enforce many type-specific privacy policies. Such policies are applied uniformly to all data of a specific type in Bing's back-end [32]. Thoth [17], on the other hand, additionally supports individual policies for each data source by integrating a compliance layer into the Linux kernel, thus enforcing both type-specific and individual policies.

Although large part of the problem has been solved by policy enforcement systems such as Thoth [17] and Grok [32], coming up with a correct policy that allows all and only compliant flows is a challenging and labor-intensive task.

Designing a correct policy that meets all specified requirements is a key step towards having privacy preserving systems. In this work, we refer to a policy as the full specification of all privacy and integrity rules in effect for the data it gets attached to. The policy designing process requires careful balancing of privacy and utility; a policy can be to permissive that it may allow illegitimate flows, similarly it can be too restrictive that it may deny legitimate flows.

A correct policy enforcement requires that all specified policies on data sources should be enforced at the system boundaries or data sinks (e.g. network socket). Specifying data sources and sinks policies can be overly strict because a policy may be progressively relaxed throughout the execution. This policy relaxation can be due to a satisfied condition that allows declassifying some data to an internal conduit (such as: files, pipes, and network connections). Designing system internal conduits consumes a lot of time because each internal conduit,in addition to sources and sinks, should be inspected whenever a legitimate flow is denied in order to identify the reason behind this flow denial.

PolSim simulates the policy flow based on data flow graphs which would help address the previous challenges. However, building such a simulator incurs its own challenges:

1. Simulating policies designed for runtime environments, how to make up for the missing runtime information? (tackled in Subsection 3.1.1)

2. Identifying what kind of policy abstractions can be introduced; abstractions that would make modeling the simulation more intuitive, and still allow for effective static analysis (tackled in Subsection 3.1.1)

3. In a runtime environment, sessions get authenticated with multiple keys, many flows are happening and processes deliver files based on runtime data. In a simulator, it is impossible to model thousands of users and millions of searchable documents. Modeling a representative set of users and files with a set of policy templates and session keys may be doable but is very labor-intensive (tackled in Section 4.2)

4. How to present the simulation results and debugging hints in a human readable format (tackled in Subsection 3.1.5)

PolSim does **meta-data (policy) flow simulation**; policies are written in a precise and declarative language (the Thoth policy language [17]) and the flow is specified in the form of a data flow graph describing how data flows between files and processes of the system.

Similar languages could have also been used, nevertheless, we use Thoth's policy language as it allows specifying declassifications [31] in addition to the standard access control policies. Thoth's policy language also supports declassifications based on data types. Finally, choosing Thoth's policy language would facilitate validating the policies deployed in Thoth as a case study.

PolSim is a testing tool; it's coverage is not complete. PolSim can be used in conjunction with policy enforcement systems (such as Thoth) to help verify the compliance of the source and sink policies by simulating the policy flow and enforcement. In addition, PolSim suggests internal conduit policies based on the provided data flow. The suggested policies can be reviewed by the policy designer if needed. PolSim approximates how the flow will behave in a deployed system given the set of policies provided as an input.

PolSim can be further developed to statically analyze deployed systems' data flows as we discuss in the outlook, Section 5. Up to our knowledge, there are no similar systems that do policy flow simulation to detect and report policy violations.

1.1 Outline

The rest of this paper is structured as follows: In Section 2, related work is discussed followed by important background information about Thoth and its policy language which is crucial to understanding the rest of this paper. In Section 3, an overview of PolSim and how it works is given. Section 4 shows a use case, how coverage testing is done, and a short performance overview. We conclude with a discussion of the scope of our tool and the outlook in Section 5.

2. BACKGROUND

2.1 Related work

2.1.1 Data flow simulation

Simulation of the data flow when a program or system is complex can be beneficial to reason about the intended behavior of the program/system. For example, a processor designer finds it useful to simulate certain instruction runs on the designed Instruction Set Architecture as a step in validating their design [33, 29]. So the goal of this kind of simulation is to validate designer's expectations about the correctness criteria that are expected of their product.

Performance goals are another broad set of goals that a simulation of a system may help achieve as in [18]. The simulation in this case needs to perform measurements that are based on the specific implementation details (unlike validation of correctness criteria where the simulation generally abstracts away implementation details).

Simulators can also be helpful for illustrative purposes. For example, studying a complex and interconnected hardware architecture may be made easier through experimenting with a simulator like the MARS simulator [33], to which an alternative would be to either buy/implement the actual hardware (which can be costly) or to study the full specifications that are provided by the designers (which can be time-consuming).

The taxonomy mentioned above does not mean that these kinds of goals are orthogonal. For example, in the domain of communication and networks, all goals may be relevant. A networks protocol designer can make use of as simulator like ns-2 [2] to measure how a network congestion may affect the speed of data transmission [4]. However, ns-2 is also routinely used in educational courses to help students understand/visualize subtle implementation details of a given network protocol.

In our work, in the context of security policy design, the development of a tool like PolSim helps system administrators by simulating data flow within a computer system so as to help them validate a proposed policy. Also, PolSim can help identify illegitimate access to resources that should be otherwise denied by the security policy under design.

2.1.2 Information flow control

Confidentiality and integrity are growing concerns for data processing systems users [7, 6, 1, 3, 8, 10, 9, 34]. Information flow control [22] is one of the techniques to preserve confidentiality and integrity requirements described in policies. Policies can be enforced through information flow control at different levels.

Policy enforcement can be done in the type system of the programming language as explained in [30]. Jif [27], for instance, augments Java with information flow types; this can be used in role-based [27] and purpose-based [21] flow restrictions. Policy enforcement can also be achieved by means of runtime checks (Resin [35], Nemesis [14]) that rely on manual application level data assertions, or by means of mandatory access control checks implemented in language libraries (Hails [20], e.g., implements security policies for the Model-Viewer-Controller (MVC) architecture).

Static analysis like in (Grok [32]) is one useful technique that guides policy enforcement. Grok uses information flow analysis to ensure policy compliance. It maps data types in code to high level policy concepts (written in their policy specification language, LEGALEASE). This is achieved by means of heuristics and selective manual verification done by developers to assign abstract labels to data files. Once this mapping is established, policy compliance checking can be performed by means of static data flow analysis.

OS kernel-level policy enforcement mechanisms are also widely studied (e.g., Asbestos [16], HiStar [36], Flume [23], Silverline [26], Thoth [17]). There are also other techniques such as software fault isolation (duPro [28]) or hypervisors (Neon [37]) that achieve security policy enforcement goals.

Thoth differs from other systems mentioned above in a number of ways; for instance, unlike language-based information flow control, Thoth applications can be written in any language. In contrast to enforcement mechanisms in OS kernel that tracks abstract labels derived from the access policies, Thoth track and enforces the declarative policies attached to data [17]. Thoth decouples policy specifications from application code which means that simulation of policies can be done independently using an external tool such as PolSim.

The Thoth policy language is based on Datalog and linear temporal logic (LTL). Datalog and LTL have been well-studied as foundations for policy languages as in [24, 13, 15] and in [11, 12, 19], respectively. Such languages are known to be clear, concise, and support high-level of abstraction [17].

2.2 Policy compliance with Thoth

As described by its creators, Thoth is "a kernel-level policy compliance layer that helps the provider of a data retrieval system enforce confidentiality and integrity policies on the data it collects and serves". Thoth does process-level information flow tracking and policy enforcement based on declarative policies attached to data. Thoth tracks data flows by intercepting all IPCs and I/O in the kernel and propagates the policies along data flows. Policies are enforced when data leaves the system or gets declassified.

In the next sub-subsections we give a summary of what Thoth is, an overview of its policy language, and some background information as explained in the Thoth paper[17] and its technical report [5].

2.2.1 Thoth architecture and design

Thoth is a distributed system that does coarse grained (processes-level granularity) information flow tracking and policy enforcement based on declarative policies attached to data.

Thoth nodes comprise three *trusted* components (in addition to the entire OS kernel and everything it depends on). A kernel module for intercepting inter- process calls and I/O requests, a reference monitor that maintains process taint sets (Explained in Sec.2.2.3) and evaluates policies, and a persistent store for meta-data and transaction logs. A global policy store is a fourth *trusted* but centralized component that can be accessed only by the reference monitors. The global policy store can be replicated at every node, provided that it provides a consistent view of policies.

In Thoth, processes (tasks) are either `CONFINED` or `UNCONFINED`. A confinement boundary separates the system core processes and unconfined processes that represent external users or components. Communication between processes is done via conduits. Processes are not trusted by Thoth, and there are no assumptions about the nature of bugs in application code. A `CONFINED` process can read any conduit with no exceptions. However, Thoth enforces the policies of all read conduits whenever a `CONFINED` process tries to write to any conduit.

Conduits that cross the confinement boundaries, have ingress (inbound conduit) and egress (outbound conduit) policies. In Thoth, all processes start `UNCONFINED`, a process can change its state to be `CONFINED` through a Thoth API call, but not the other way around, to prevent a process (task) from reading data in the confined state and leaking the data without any policy protection in the `UNCONFINED` state.

2.2.2 Thoth policy language overview

Thoth policy language is *"a declarative policy language that supports confidentiality, integrity, provenance, and declassification policies that may reference client identity, time, current content, proposed content in a transactional update, history, and data flow"* [17].

Thoth's policy language can be used to specify the policies of conduits. Access to a conduit's data and data derived from it should comply with the attached policy. It specifies the rules for protecting data integrity and confidentiality. A conduit policy is a two-layer specification: access control and declassification. In the first layer, a conduit's **read** and **update** rules are specified, i.e., who can read or update the conduit and under which conditions (e.g., only Alice after a certain date). In the second layer, a **declassify** rule restricts the access policies downstream (i.e. the access policy on derived data). It also allows declassifying data by allowing the policy to be relaxed progressively as stipulated conditions are met.

The policy language allows expressing content-dependent policies through what is termed "typed declassification". Typed declassifications allows declassifying data based on the content type, e.g., declassify data only if the data entries are valid file names in the system. A condition is expressed as a set of predicates (Table 1) connected with conjunctions ("and", written $\wedge$) and disjunctions ("or", written $\vee$). Policies are written as rules, each written in the form "(**perm** :- `cond`)", which means the permission **perm** is granted if some condition `cond` is satisfied. For example, the rule "**read** :- true", stipulates that the read permission is granted to everyone, i.e. any one can read the conduit.

We consider an example of an indexing and search pipeline, which was discussed in Thoth. The search engine indexes clients' private, semi-private data and public web content. A basic security requirement would be that private data (email, calendar, profile personalization, etc.) should be visible and editable only by the client. Semi-private data should be editable by the client and readable by his friends for instance. Public data can be read by anyone.

This can be expressed using the Thoth policy language as follows, assuming that the private data is for the client *Alice*:

- **Private data**
  - **read** :- sKeyIs($k_{\texttt{Alice}}$)
  - **update** :- sKeyIs($k_{\texttt{Alice}}$)
- **Semi-private data**
  - **read** :- sKeyIs($k_{\texttt{Alice}}$) $\vee$ (sKeyIs(K) $\wedge$ ("Alice.acl", `Offset`) says isFriend(K))
  - **update** :- sKeyIs($k_{\texttt{Alice}}$)
- **Public content**
  - **read** :- *true*
  - **update** :- sKeyIs($k_{\texttt{Alice}}$)

The predicate sKeyIs($k_{\texttt{Alice}}$), means that the active session is authenticated with Alice's public key, denoted $k_{\texttt{Alice}}$.

In semi-private data access, only Alice or a friend of Alice can read the content. This can be achieved by allowing read access only to an active session authenticated with Alice's public key **or** a key of a client who is a friend of Alice. This condition can be interpreted as follows: the key `K` that authenticated the current session exists in the trusted key-value tuple Alice.acl, which represents Alice's friends list. This can be extended to support friends of friends privacy requirements.

Arithmetic/string		**Conduit**		**Content**	
add(x,y,z)	x= y+z	cNameIs(x)	x is the conduit pathname	(c,off) says $(x_1,\ldots,x_n)$	$x_1,\ldots,x_n$ is the tuple found in conduit c at off
sub(x,y,z)	x= y-z	cIdIs(x)	x is the conduit id		
mul(x,y,z)	x= y*z	cIdExists(x)	x is a valid conduit id	(c,off) willsay $(x_1,\ldots,x_n)$	ditto for the update of c in the current transaction
div(x,y,z)	x= y/z	cCurrLenIs(x)	x is the conduit length		
rem(x,y,z)	x= y%z	cNewLenIs(x)	x is the new conduit length	each in (c,off) says $(x_1,..,x_n)$ {condition}	for each tuple in c at off, assign to $x_1,..,x_n$ and evaluate condition
concat(x,y,z)	x= y||z	hasPol(c, p)	p is conduit c's policy		
vType(x, y)	typeof(x)=y?	cIsIntrinsic	does this conduit connect two confined processes?	each in (c,off) willsay $(x_1,..,x_n)$ {condition}	ditto for the update of c in the current transaction
Relational		**Session**		**Declassification rules**	
eq(x,y)	x=y	sKeyIs(x)	x is the session's authentication key	c1 until c2	condition c1 must hold on the downstream flow until c2 holds
neq(x,y)	x!=y				
lt(x,y)	x<y	sIpIs(x)	x is the session's source IP address	isAsRestrictive(p1,p2)	the permission p1 is at least as restrictive as p2
gt(x,y)	x>y				
le(x,y)	x<=y	IpPrefix(x,y)	x is IP prefix of y		
ge(x,y)	x>=y	timeIs(t)	t is the current time		

Table 1: Thoth policy language predicates and connectives

A **declassification** rule controls the policies on downstream conduits, written also in the form (**declassify** :- cond), where "cond" is a condition on all downstream sequences of conduits. **Declassification** rules may additionally contain the connective (c1 `until` c2), which means that condition c1 holds on every conduit downstream `until` the condition c2 holds (no checks are needed beyond that). For instance, "cond" may stipulate that the conduit is readable only by Alice `until` the current year is at least 2017. This means, as long as 2017 is yet to come, only Alice can read this conduit. Starting 2017 every one can read it. In addition to the `until` connective, the predicate isAsRestrictive(`p1,p2`) can be used, which checks that permission `p1` is at least as restrictive as permission `p2`.

A search engine generates an index file computed from the whole data corpus (including private data of more than one client), this would result in a non readable index; because the policy on the index would be overly restrictive. For the search pipeline to work, some data from the index file should be declassified (i.e. file IDs (conduit IDs) in case of a search engine, as they will form the search results afterwards). To allow for this declassification, all searchable content must allow for that declassification. This can be achieved by adding the following declassification rule to every searchable data item:

> **declassify** :-
> isAsRestrictive(**read**, *this*.**read**) until `ONLY_CND_IDS`

This rule stipulates that data derived from Alice's data can be written into a conduit which satisfies the macro `ONLY_CND_IDS`. A *macro* is a special condition that packs multiple predicates and connectives into a single entity. A *macro* gets expanded only during the policy evaluation (explained in section 2.2.4). Macros make the policies more readable and easier to understand. The macro `ONLY_CND_IDS` expands to

> cCurrLenIs(`CurrLen`) ∧ cNewLenIs(`NewLen`) ∧
> eachin(this, `CurrLen`, `NewLen`) says(`Cndid`)
> {cidExists(`Cndid`)}

This stipulates that every entry in the newly written data is a valid conduit id. In other words, only conduit ids can be declassified without further constraints. If we additionally assume that the conduit ids are themselves confidential, we need to make sure that conduit ids are declassified to conduits having a policy as restrictive as the policy of each of the conduits whose conduit ids where declassified. Hence, the `ONLY_CND_IDS` macro can be modified to:

> cCurrLenIs(`CurrLen`) ∧ cNewLenIs(`NewLen`) ∧
> eachin(this, `CurrLen`, `NewLen`) willsay(`CndId`)
> {cIdExists(`CndId`) ∧ hasPol(`CndId` , `P`) ∧
> isAsRestrictive(**read**, `P`.**read**) ∧
> isAsRestrictive(**declassify**, `P`.**declassify**)}

In this macro expansion, hasPol(`CndId`, `P`) binds `P` to the policy of the conduit `CndId`, and the predicates isAsRestrictive(**read**, `P`.**read**) and isAsRestrictive(**declassify**, `P`.**declassify**) require that the read and declassify rules of the conduit satisfying this macro are at least as restrictive as those of the conduit with conduit Id `CndId`. This modified macro is named `ONLY_CND_IDS+`.

More data retrieval policies can be found in the Thoth paper[17] and its Technical Report [5].

2.2.3 Idea of taint tracking

A taint is meta-data attached to data, the taint flows with the data. Thoth introduced the *policy as a taint* idea. The entities that get the taint (policy) attached to are processes (tasks) and conduits. The operation that potentially causes update/attachment of a new taint is reading or writing a conduit.

In data retrieval pipelines, processes read from and write to conduits. Whenever a process reads a conduit, the process gets tainted by the conduit's policy. Processes keep getting taints as they read from conduits; this forms a taint set. All policies in the taint set are checked and respected whenever a conduit is written or data crosses the confinement boundary.

Taint set policies are checked whenever any of the aforementioned conditions apply, because there is a potential data flow from every conduit that a process p has read in the past to the conduits p might write to in the future. This ensures that data will flow to a conduit f only if:

- either f has a policy as restrictive as all policies of source conduits (policies in the taint set of p),
- or the data can be safely declassified to f.

2.2.4 Thoth policy enforcement algorithm

When a process accesses a conduit, the checks made by Thoth depend on whether the process is `CONFINED` or `UNCONFINED`. As shown in the policy enforcement algorithm in Figure 1, in `CONFINED` processes, whenever a conduit is read, its **declassification** rule is added to the taint set of the reading process. However, when a conduit is being written, all **declassification** rules (c `until` c') must hold in one

Question: Should a conduit read or write be allowed?

```
Inputs:   t, the task reading or writing the conduit
          f, the conduit being read or written
          op, the operation being performed
               (read or write)
Output: Allow or deny the access
Side-effects: May update the taint set of t
1  if t is UNCONFINED:
2    if op is read:
3      Check f's **read** rule.
4    if op is write:
5      Check f's **update** rule.

6  if t is CONFINED:
7    if op is read:
8      Add f's policy to t's taint set.
9    if op is write:
10     //Enforce declassification policies of t's taint set
11     for each **declassification** rule (c until c')
       in t's taint set:
12       Check that EITHER c' holds OR (c holds AND
         f's declassification policy implies (c until c')).
```

Figure 1: Thoth policy enforcement algorithm

of two ways: either c' holds, thus data can be declassified, or c holds and the conduit being written to enforces (c `until` c').

To check if a policy is implied and to apply the isAsRestrictive(p1,p2) checks, policies have to be compared. This is done using the following heuristics:

1. **Equality**: Comparing hashes of both policies, i.e checking if both policies are identical.

2. **Inclusion**: Making sure all predicates in the less restrictive policy appear in the more restrictive policy taking the conditions structure (variables, conjunctions, and disjunctions) into account. This can be checked as follows:

   - Predicate b is at least as restrictive as predicate a, if this is known a priori; e.g., b is false.
   - Conjunction b is at least as restrictive as conjunction a (isAsRestrictive(b, a)) iff:
     $\forall$ predicates x in a . $\exists$ a predicate y in b : isAsRestrictive(y, x).
   - Disjunction b is at least as restrictive as disjunction a (isAsRestrictive(b,a)) iff:
     $\forall$ conjunctions x in b. $\exists$ a conjunction y in a : isAsRestrictive($x.y$).

3. **Partial evaluation**: If not all **declassification** rules (c `until` c') in a process's taint set hold, the holding ones are dropped and the rest have to be implied by the **declassification** rule of the conduit being written to, this can be checked using Equality and Inclusion.

   Partial evaluation allows for declassifying data progressively through the data processing pipeline. This can be seen as a multi-step declassification. For example, if a file policy stipulates that it can be only declassified to a file allowing a list of file names to be written **and** having a policy isAsRestrictive(**declassify**,P) Partial evaluation will allow such policy to propagate if the data written to a file is a list of file names and the file have a policy that implies isAsRestrictive(**declassify**, P).

3. POLSIM SYSTEM

Designing policies for data retrieval systems requires a deep understanding of all possible flows in the underlying data processing pipeline. Lax source and sink policies may not prevent data leaks. Similarly, an overly restrictive source or intermediate policy may hinder work flow by blocking legitimate data flows.

PolSim is a static analysis tool for rapid policy-aware system simulation. It simulates the data policies flow of a dynamic system given the source policies, optional intermediate policies, and a partial order on the expected data flow. PolSim can simulate all possible data flow scenarios and give the policy designer a summary of the potential data flow violations caused by the (possibly overly-strict) source or intermediate policies.

PolSim does not require the source code of the system under simulation, nor actual runtime data (but it needs an abstraction of the policy-relevant subset of the latter). PolSim uses the Thoth policy language together with few annotations and hints provided by the policy developer to simulate the flow of policies, generate intermediate inferred policies and report any non-compliant flows.

3.1 System overview

As shown in Figure 2, PolSim work flow is a four step sequential process:

1. Simulation environment construction:
   (a) A system description file is parsed and an Abstract Syntax Tree (AST) is formed.
   (b) A simulation environment (that maintains the data structures required to perform the simulation), is constructed while walking the AST.

2. The flow simulator simulates the data flow and the policy enforcement mechanisms.

3. The policy violation reporter investigates the causes of flow denial, if any.

4. The simulation report is constructed based on information from the flow simulator and the violation reporter is used to construct the simulation report.

In the following sub-subsections, we explain each step of the work flow thoroughly.

3.1.1 Simulation environment construction

Simulation environment construction is the process of transforming the written system description into a simulation environment that maintains all policies and configurations needed for the simulation.

The user provides as input to PolSim a directed graph, whose nodes represent the processes and conduits in the system, and whose edges represent *potential* flows (if there is no edge from A to B, no flow is possible from A to B). This graph is a very coarse abstraction of the actual system, since it includes no source code and no actual conduit data but, with one minor exception explained later, this is all that PolSim needs. With each ingress conduit, the user specifies a Thoth policy. Policies may also be optionally specified for any other conduits. Similarly, missing runtime information (such as IPs, current time, sample data) can also be specified for any conduit. Next, the user provides a flow description, which is a sequence of edges depicting the order in which flows are expected to occur during actual execution. In practice, one way to generate the data flow graph is to

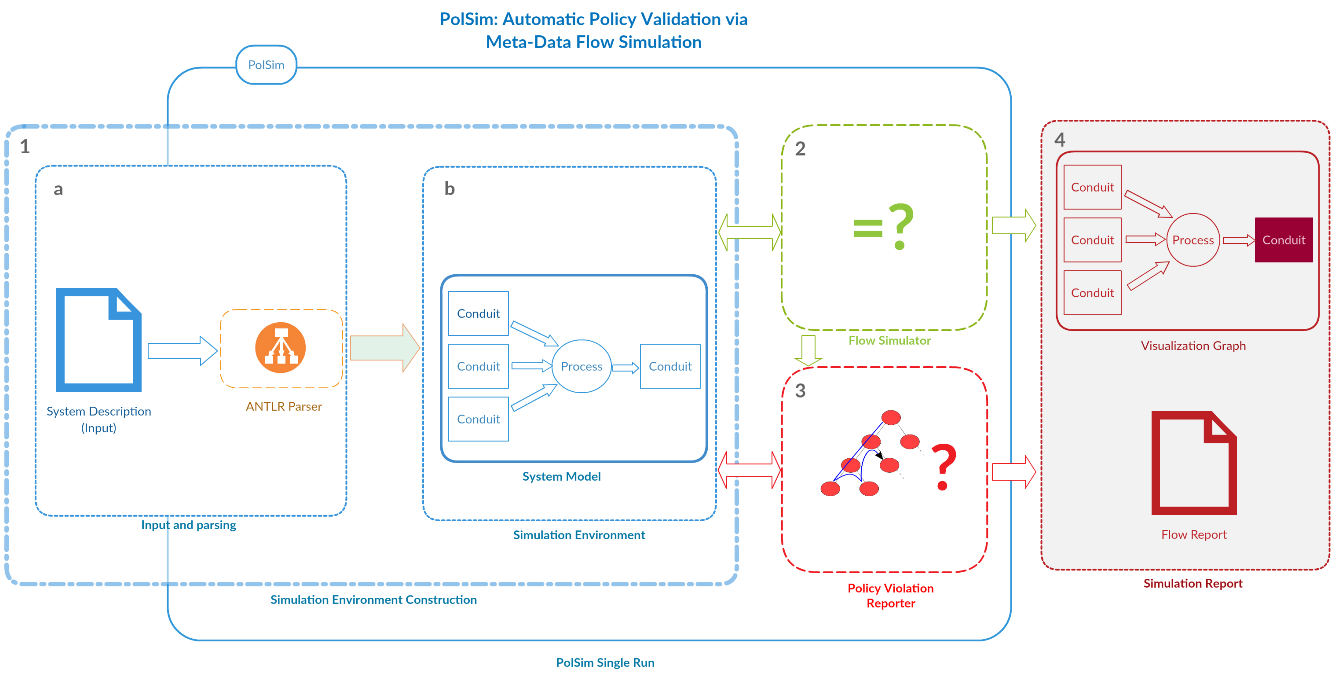


Figure 2: PolSim work flow

record actual IPCs and file reads and writes in a production system before policies have been added. This is briefly discussed in Section 5.

User defined predicates and their relations

The Thoth policy language features a set of predicates (Table 1) that are used in declaring data privacy policies. To allow for further policy abstractions, we introduce user defined predicates which allow policy developers to specify generic predicates with a predefined number of arguments. This gives the policy designer the flexibility to chose a representative name for what a predicate or a macro may be doing in the system under simulation. It also allows the policy designer to abstract some of the macros as predicates. This is useful when a macro expansion is irrelevant or not the focus of the current flow being simulated (e.g., the policy designer assumes the macro is correct and its expansion will evaluate to true; thus can be abstracted and compared as a unit without expanding and evaluating it). As we show in our use case in Section 4, a system that has many semi-private policies, a FriendsOf(x) macro can be introduced and expands to:

$$\mathsf{FriendsOf}(\mathtt{x}) = \mathsf{sKeyIs}(k_{\mathtt{x}}) \vee$$
$$(\mathsf{sKeyIs}(\mathtt{K}) \wedge \mathsf{concat}(\mathtt{L,\ x,".acl"}) \wedge$$
$$(\mathtt{L,\ Offset})\ \mathsf{says}\ \mathsf{isFriend}(\mathtt{K}))$$

Expanding this macro during the simulation may be irrelevant because we are only interested in knowing that some file is only accessible to Alice and her friends; thus we can define FriendsOf as a predicate instead.

In addition, any two generic predicates may have a restrictiveness relation as generic as p(x,y) << q(y,z) which means that predicate p is more restrictive than predicate q provided that p's second argument is identical to q's first argument. Specifying that FriendsOf("Alice") << SKeyIs("Alice") would permit the data flow from a policy that has a permission stipulating FriendsOf("Alice") to a conduit with a policy stipulating SKeyIs("Alice"), but not the other way around.

User-defined predicates contribute to solving challenge number two as discussed in subsection 1.

3.1.2 *Simulation environment*

The system description file is read, and parsed to generate an abstract syntax tree. The simulation environment maintains the data-structures required to perform the simulation, it contains the data flow graph, defined macros, user-defined predicates, restrictiveness relations between user defined predicates, variable bindings, status of the simulation (compliant flows so far, and *processTaintSet*). The data flow graph consists of nodes and edges. Nodes represent conduits and processes, while directed edges represent the flow of data between nodes.

3.1.3 *Flow and policy enforcement simulation*

Figure 1 summarizes the abstract checks that Thoth does. PolSim's algorithm is a simplification of Thoth's algorithm: We do not simulate flows to/from UNCONFINED processes, and conditions are checked on the abstract system state, rather than concrete data.

The enforcement process starts by simulating the flow of policies while respecting the total/partial order of flows provided by the policy designer. As in Thoth, declassification rules are the key players when it comes to enforcing policies. The reason behind this is that declassification rules allow data items to be revealed (declassified) unconditionally or under some reasonable conditions. Declassification rules, when used correctly, prevent the data processing pipeline from getting blocked due to process over-tainting.

PolSim simulates the policy flow in the CONFINED region and its boundaries only, since this is the only relevant region where declassifications have to be enforced. Since our tool does not deal with the runtime environment, we do not have explicit reads or updates. Read and update rules are checked only if referred to during declassification rules.

In order to make our terminology clear we would like to clarify two important terms: a ***compliant flow*** is a flow that is compliant with the ingress and egress policies, i.e., a

flow that doesn't cause data leakage.
a ***flow block*** is what happens when a policy violation is detected, whether it was expected or not.

Thoth optimizations are also applied in PolSim such as allowing for partial declassification and taint compression (a policy is not added if the taint set already includes an equally or more restrictive policy). In our simulations, data is declassified at the earliest point in the pipeline to allow as many flows as possible.

Intermediate conduits policy generation

To reduce the policy developer's burden, PolSim suggests policies of intermediate conduits automatically if they are not explicitly set by the policy developer. PolSim generates a policy that is as restrictive as all policies in the taint set of the process that writes the conduit after applying relevant declassifications. Moreover, the generated policy access control rules are set accordingly to satisfy all the restrictiveness requirements of the form (isAsRestrictive(**read**,p1) and isAsRestrictive(**update**,p2)) of the writing process's taint set. This helps the policy designers ignore all intermediate conduit policies and only investigate the intermediate conduits' generated policies when the data flow is blocked.

3.1.4 Policy violation reporting

Whenever a flow is denied due to a non compliant policy, the flow validation process terminates and reports that a flow blocked at conduit c in the pipeline due to some unsatisfied predicates. At this point, the policy violation reporter becomes in charge of identifying the reasons behind this flow blockage. As defined before, a flow block is what happens when a declassification fails and propagating the policy forward is impossible, i.e data breach may happen if the data flow continues through the pipeline. A blocking predicate/-macro is the predicate or macro that didn't evaluate to true or didn't propagate successfully. Policy violation reporter makes use of the following

1. The maintained meta-data about the origin of the blocking predicate
   A reference to the conduit that had this predicate as a part of its policy specification, and thus was propagated through the data processing pipeline.

2. The list of blocked predicates
   Once a blocking predicate has been identified, it is added to the set of blocking predicates. Given multiple disjunctions, if all disjuncts cause blocking, then the first blocking predicate of each disjunct will be the one reported (this is only a heurestic, that can be changed).

Once these pieces of information are gathered, the policy violation reporter tracks the flow blocking path by means of depth first search. The DFS starts from the violating predicates' original policies till it reaches the conduit where the blockage happened, and the path between them is reported. The policy violation reporter ensures that the tracked path is correct by checking that the predicate existed and had the same predicate ID in that path. This is important because a predicate of the same type may have existed in two different branches of the data flow.

3.1.5 Simulation reports

Validating the policy requires making sure that all code executions that should flow do indeed flow and that all code executions that shouldn't flow are denied (blocked).

This raises the need to have a detailed human readable report explaining how the simulation went. Accordingly, PolSim reports:

1. A visualization of the system as a graph that shows:
   (a) `CONFINED` and `UNCONFINED` regions
   (b) Conduits and Processes
   (c) Explored paths
   (d) Conduits with automatically generated policies
   (e) The conduit where the flow got blocked
   (f) Blocking macros and predicates
   (g) A trace of all blocking predicates paths

2. A report about the current flow run that shows:
   (a) A summary of the current run
   (b) Conduit policies
   (c) Suggested intermediate conduit policies
   (d) Blocking conduit policy
   (e) Process taint-sets
   (f) Successful declassifications
   (g) Detailed information about blocking macros and predicates

Simulation reports contribute to solving the fourth challenge discussed in subsection 1.

3.2 Summary of abstractions in PolSim

PolSim aids system policy developers by allowing them to rapidly prototype system policies and simulate flows. Intermediate conduit policies are suggested and flow blocks are reported. Flow block is what happens when a declassification fails and propagating the policy forward is impossible, i.e. a data breach will happen if the data flow continues through the pipeline. PolSim tries to approximate the runtime environment so as to give the policy designer a precise view of how compliant the data flow will be, given the policies described.

PolSim implements the same predicates as the Thoth policy language (Table 1) in addition to *generic predicates* (predicates with predefined number of arguments that get compared syntactically). These predicates allow systems designers to abstract macros and make the model more intuitive and easier to understand and debug.

PolSim does not need the actual implementation of the system nor the runtime data. It just needs the data flow graph, conduits and processes involved, source and sink conduit policies and hints about the data when a policy is content-sensitive (in the form of conduit and system state in the system description file).

4. EVALUATION

To demonstrate PolSim features, we show, as a use case, how search engine policies can be developed using the help of PolSim. PolSim, as we show, allows fast prototyping of the system as there is no dependency of requiring an actual system to test the policies against. Developing such policies without a simulator would be much harder. The policy developers have to manually inspect and identify possible blocks and their causes.

4.1 Use cases

In this section, we simulate policy flow in a data processing pipeline that was discussed in Thoth. Simulating the final policies approved by the policy designers would show that the pipeline never gets blocked. However, we are interested in showing how such a tool can make developing a policy that allows all compliant flows much easier. As an input to PolSim we need the following:

- Conduits and Processes involved in the simulation
- Source conduits policies
- Partial order on the flow (a data flow graph)

Further, through the policy development iterations we modify the intermediate conduits policies when needed.

4.1.1 Thoth search pipeline

Consider the indexing and search pipeline discussed in Thoth:
In a search engine pipeline, there are files, pipes, network sockets and processes. Each of the files have it's own policy. The Search pipeline looks as follows:

The search pipeline as shown in Figure 3 works as follows

- Indexer process indexes all searchable files in the system
- Indexer writes the index file for later lookups
- End user (Alice) authenticated with a session key ***K*** connects to the network socket
- The end user issues a query to the front end
- The front end forwards the query to the search process
- The search process looks up the index
- The search process writes the search results in the search results file
- The front end reads the search results file, fetches relevant files from the system extracts snippets and writes back the query results to the network socket

For the sake of this simulation we will assume that the searchable corpus contains files having three policies.
The input policies are:

- **Private data**
  - **read** :- sKeyIs(k)
  - **update** :- sKeyIs(k)
- **Semi-private data**
  - **read** :- FriendsOf(k)
  - **update** :- sKeyIs(k)
- **Public content**
  - **read** :- *true*
  - **update** :- sKeyIs(k)

We define SKeyIs(k) $<<$ FriendsOf(k) which means that SkeyIs with argument k is at least as restrictive as FriendsOf when the same argument K is used. Moreover, each file has a default **declassify** rule which stipulates that data cannot be declassified (the declassification condition is false):

$$\textbf{declassify} \;\text{:-}\; \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read})\ \texttt{until}\ \mathsf{false}$$

For this use case, the searchable corpus consists of four files, two of which have a private policy for Alice and Bob. A third file, AlicesFriends, has a semi-private policy. There is a publicContent file with a public policy.

Running PolSim with this input would result in the following simulation summary, Figure 4

In this simulation, PolSim simulated the flow of policies based on the given input. Files were indexed by the indexer, the search process looked up the index file and wrote the search results, the front end read the search results, fetched the relevant files and forwarded them to the network socket authenticated by Alice. At this point, the flow got denied. What actually happened was that PolSim propagated the policies of ingress conduits forward, until it was impossible to continue the simulation (reaching the network socket with a non compliant policy).

The propagated policy is more restrictive than all ingress conduits policies as it is generated from the Indexer process taint set. This policy doesn't allow data flow to any conduit, simply because the indexer consumes data from multiple sources, some of which have private data policy for different users.

The system tried to declassify the data to the network socket, but declassification failed due to the predicate false in the until part of the **declassify** rule. Even propagating the policy failed because the network socket is authenticated by Alice and the front end's taint set include a **declassify** rule that can not be propagated except to a conduit having a **read** rule as restrictive as SKeyIs(Bob).

Thanks to PolSim, the policy designers can clearly see the problem, the front end has Alice's and Bob's private policies in the taint set and is trying to write to the network socket authenticated by Alice. Bob's policy must have reached the front end through the search results (as highlighted in the graph). The search results conduit had no defined policy initially, thus PolSim suggested a policy that is as restrictive as the search process taint set; so as to allow the data flow to continue. Similarly, the index file had no defined policy initially, thus PolSim suggested a policy that is as restrictive as the indexer process taint set.

A solution to this problem can only come from the policy developer who understands the actual system data flow. The policy developer knows that the search process writes the search results as a list of URLs (conduit IDs). So a potential fix to the current policy (also adopted by Thoth) is to allow the index file to drop the taint from conduit IDs (i.e. declassify them).

This can be achieved using the `ONLY_CND_IDS` macro which allows declassifying conduits' names only. `ONLY_CND_IDS` as described in the introduction expands to:

$$\begin{array}{l}\mathsf{cCurrLenIs}(\texttt{CurrLen}) \land \mathsf{cNewLenIs}(\texttt{NewLen}) \land \\ \mathsf{each\ in}(\mathsf{this}, \texttt{CurrLen}, \texttt{NewLen})\ \mathsf{says}(\texttt{Cndid}) \\ \quad \{\mathsf{cidExists}(\texttt{Cndid})\}\end{array}$$

which stipulates that the written data are IDs of existing conduits in the system.
We can actually specify this **declassify** rule in one of two ways:

$$\begin{array}{ll}\textbf{declassify} \;\text{:-} & \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read})\ \texttt{until} \\ & \mathsf{isAsRestrictive}(\texttt{update}, \texttt{ONLY\_CND\_IDS})\end{array}$$

In this case, PolSim makes sure that the update rule of the conduit being written to (the search results) is as restrictive as `ONLY_CND_IDS`. A check of restrictiveness will be applied, PolSim will check if the update rule of the conduit that data should be declassified to is as restrictive as `ONLY_CND_IDS`. Once the until condition is satisfied, the evaluation of the

[1]as generated by PolSim
[2]graph optimized for printing by referring to the blocking predicates outside the graph

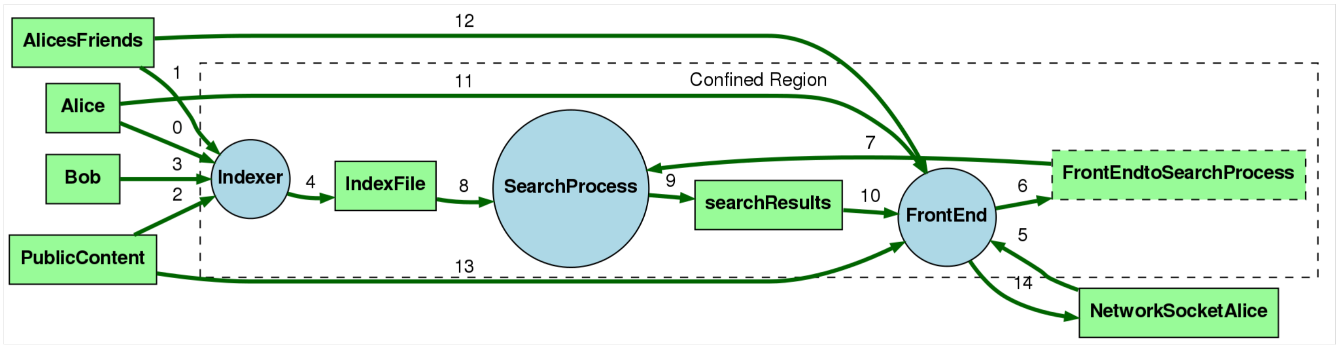
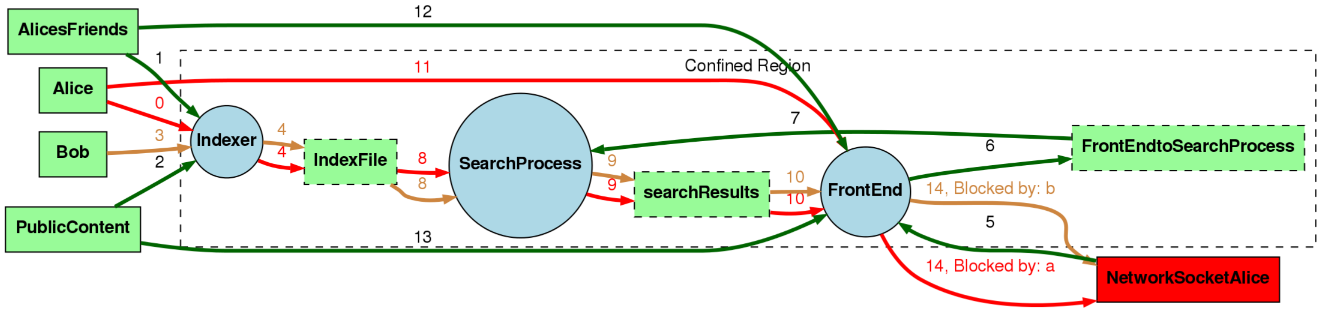


Figure 3: Search engine data processing pipeline[1]



Blocking predicates	
a	false
b	isAsRestrictive(READ, [[SKeyIs(Bob)]])

Figure 4: First step towards developing a search engine policy[2]

macro will happen when actual data starts to be written to the conduit.

We run the simulation again, as we can see in Figure 5, there is still a problem.

This simulation reported that there were three blocking predicates: false, isAsRestrictive(**read**, [[SKeyIs(`Alice`)]]) and isAsRestrictive(**read**, [[SKeyIs(`Bob`)]]). The source files policies state that no declassification should be ever possible:

$$\textbf{declassify} \text{ :- } \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read}) \ \texttt{until} \ \mathsf{false}$$

The **declassify** rule until part will always evaluate to false; thus false was reported. Moreover, trying to propagate the whole policy failed because the (revised) IndexFile policy is more permissive than the policies of the source conduits.

Thus, all source policies have to be revised. All ingress policies should have a **declassification** rule that allows declassifying conduit IDs:

$$\begin{aligned}\textbf{declassify} \text{ :- } & \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read}) \ \texttt{until} \\ & \mathsf{isAsRestrictive}(\texttt{update}, \texttt{ONLY\_CND\_IDS})\end{aligned}$$

Also, we have to change the IndexFile's read policy to a policy that is at least as restrictive as all source files policies i.e. false:

This would make the indexer declassify rule at least as restrictive as the declassification rule in all source policies, thus the indexer will no longer be the reason behind the flow getting blocked.

As we can see in Figure 6, the indexer was allowed to index the data. The search results had no policy, thus PolSim suggested that it gets a policy as restrictive as the taint set of the search process (i.e. a policy identical to the index file policy). The flow got blocked by the network socket for obvious reasons, its update rule is not as restrictive as `ONLY_CND_IDS` thus search results can't be declassified. Moreover, the **read** rule is SKeyIs(`Alice`) which is not as restrictive as false; thus, data policies can not be propagated.

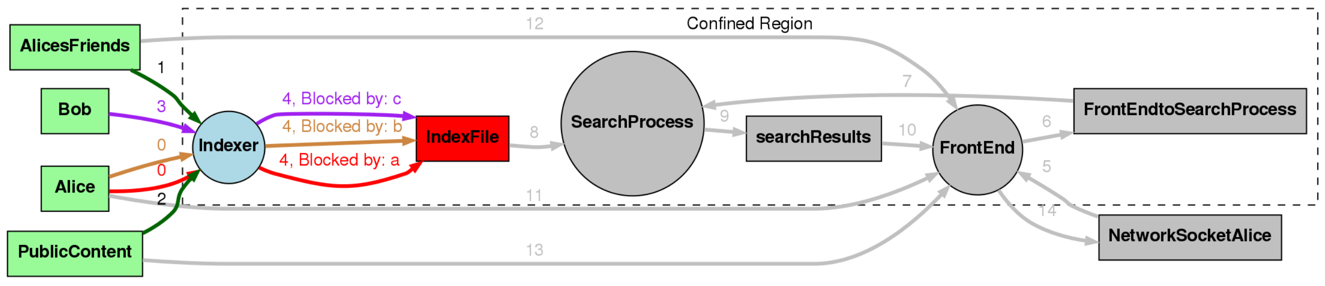
To fix this issue, we need to specify the point where conduit IDs get declassified. It is obvious that this point will be the search results conduit. Thus we modify the update rule of the search results conduit to be `ONLY_CND_IDS`.

This modification resulted in a flow block, the flow blocked due to the fact that the network socket does not have an update policy as restrictive as `ONLY_CND_IDS`. As reported by PolSim, this happened because the front end fetches the files based on the conduit IDs of the search results. This would make the front end get tainted by those files' policies. To solve this issue, we need to modify the **declassification** rules to allow declassifying the data to extrinsic conduits having a **read** rule at least as restrictive as the source policies them selves. Thus, we change all **declassify** rules of source conduits to become:

$$\begin{aligned}\textbf{declassify} \text{ :- } & \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read}) \ \texttt{until} \\ & [[\mathsf{cIsIntrinsic} \\ & \ \wedge \ \mathsf{isAsRestrictive}(\textbf{update}, \texttt{ONLY\_CND\_IDS})] \ \vee \\ & [\mathsf{Extrinsic}^{3} \wedge \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read})]]\end{aligned}$$

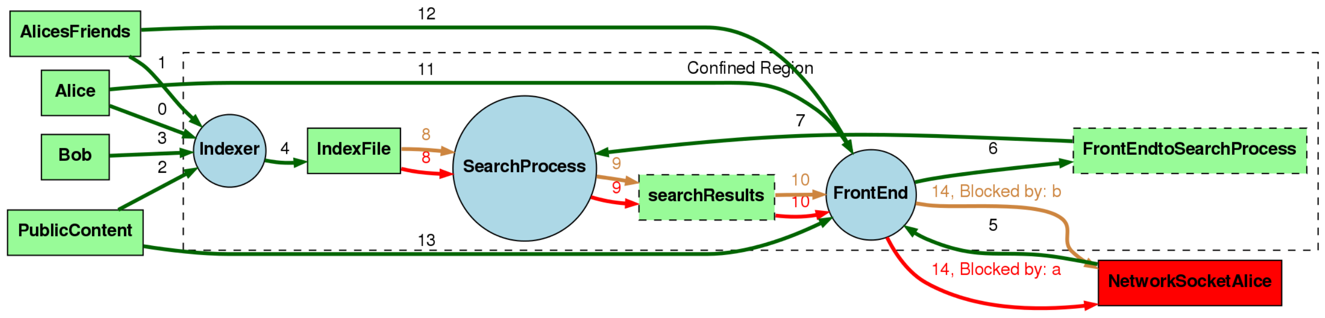
This would result in the simplest search pipeline that respects the provider's policies. The simulation would indeed succeed as in Figure 8.

[3]Extrinsic is equivalent to not intrinsic



Blocking predicates	
a	false
b	isAsRestrictive(READ, [[SKeyIs(Alice)]])
c	isAsRestrictive(READ, [[SKeyIs(Bob)]])

Figure 5: Search engine simulation with an index policy



Blocking predicates	
a	isAsRestrictive(UPDATE, [[ONLY_CND_IDS]])
b	isAsRestrictive(READ, [[false]])

Figure 6: Search engine simulation with modified ingress and index file policies

In the appendix, further development of the policy is shown, namely how can the policy be modified to support confidential conduit IDs and censorship.

We also modeled the personalization and advertisement pipeline from the Thoth paper and found no problems, discussed briefly in the thesis [25].

4.1.2 Evaluation

In this section, a basic search engine data processing pipeline has been simulated in a step by step fashion. Afterwards, the extension of the pipeline to feature common data retrieval functionality has been discussed. The tool helps the policy developer by:

- Verifying that the set of policies described allows all expected flows
- Pointing to sources of blockage upon flow blockage (due to non compliant policies)
- Pointing to the specific predicates that cause blocks

Generally the policy designer needs to modify the intermediate policies if there needs to be a specific declassification or a need to provide abstract runtime data to allow certain conditions to be met. PolSim shows the blocking predicates and their provenance which makes identifying the required modification easier than tracing the policies in a deployed system without knowing where to check for policy violations. When we started PolSim, it wasn't planned to have all the features, namely the provenance and causes of blockage. However, when we started using the tool in modeling system policies, we felt the need for such features and added them.

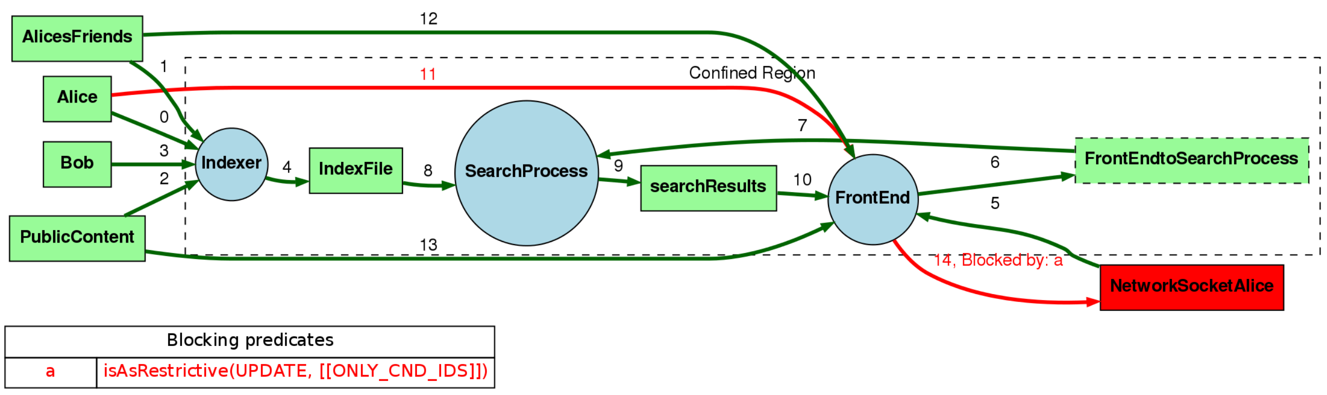
Our experience with re-developing Thoth policies from scratch indicates that PolSim is effective and useful for policy development.

4.2 Coverage testing

In the discussed example, we did not consider the case of having a faulty search process that may accidentally write Bob's conduit ID as a part of Alice's search results. Typically a policy designer will not be able to think of all possible scenarios. To address this, we introduced the concept of coverage testing by supporting multiple scenario simulation. Policy designer may want to test the system under all possible policies for source conduits, with different runs authenticated by different users. Thus, we allow policy developers to define a ***Bag Of Policies***, where the policy developer provides PolSim with all possible source policies. Our tool will try all possible permutations and assign them to all source conduits (or any conduit tagged to be assigned a random policy). This will automatically reveal cases such as Bob's data leakage. Also it gives the policy designer more insights about possible scenarios.

The policy designer gets a summary of all runs with the ability to filter through them by choosing a sub set of the blocking predicates. Policy designers can keep boiling down the problem till they find which runs to further investigate.

Coverage testing contributes to solving challenge number three from Section 1. An example of PolSim's coverage testing output can be seen in the Appendix.



Blocking predicates	
a	isAsRestrictive(UPDATE, [[ONLY_CND_IDS]])

Figure 7: Search engine simulation with modified search results policy

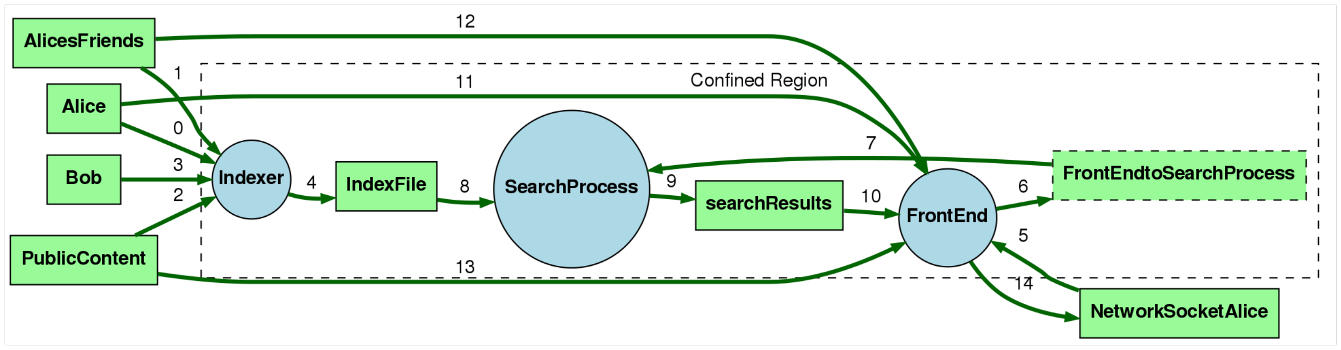


Figure 8: Minimal search engine data processing pipeline

4.3 Advanced intermediate policy suggestion

In this section we show another direct use of the *Bag Of Policies* concept. PolSim suggests intermediate conduits policies based on the taint set of the process trying to write/declassify to the conduit. However, this policy may be too strict to keep the pipeline from stalling. Usually in this case, this policy needs a custom **declassification** rule that allows the data flow to continue without revealing confidential data. Using the *Bag Of Policies* idea, the policy designer can put multiple policies with different **declassification** rules in the bag of policies and let PolSim try assigning them to this conduit and report a comparison of how the simulation went in each scenario.

4.4 Performance

PolSim is a lightweight simulation tool that works at coarse-granularity, thus there were no performance concerns. Usually a full simulation (input parsing, compilation, simulation, graph and report generation) takes less than a second. All simulations we where executed on an Intel core i5 laptop, with SSD drive and 8 GBs of RAM. Simulations were having 16 conduits and 6 processes with 33 read/write operation. Even with coverage testing, all simulations took less than a minute (16 scenarios -> 11 seconds, 32 scenarios -> 20 seconds). Of course, having thousands of scenarios will take more time.

Simulation time depends on many factors such as when the simulation stalls in the pipeline, whether the restrictiveness checks will hold, how long the logical evaluation of a condition with multiple conjuncts takes (if the first conjunct is false then the whole conjunction will yield false immediately). However, testing of multiple scenarios can be easily parallelized, hence PolSim would scale well.

5. FUTURE WORK

PolSim is a light-weight simulation tool that aids policy developers in developing their policies by simulating policy flow and reporting violations and debug information. Our tool can be further developed to have more automation and support more scenarios.

Automatic simulation of deployed systems

Given a deployed system (possibly without policies or with untested policies), a simulation description file can be automatically constructed by recording all Inter Processes Communications, conduits reads and writes in order. Recorded data can be abstracted and semi-automatically converted into a valid PolSim input.

What-if scenarios

Data retrieval systems may allow end-users to set their own custom policy, PolSim can automatically validate the added policy and check that it will not stall the pipeline. Custom policies can be automatically approved if they cause no blockage or stalls in the pipeline and do not violate company and legal mandates. This can be integrated with user-facing policy front ends.

Input range coverage testing

Having multiple scenarios simulated and compared together is a first step towards coverage testing. We would like to have the coverage testing identify ranges of specifications (time intervals, Ip ranges (countries), set of users, etc.) that would cause the pipeline to stall and ranges that would be compliant with the policies.

Better blocking predicate suggestion

Currently, PolSim reports all blocking predicates, so if a condition blocks the flow then the first predicate evaluating to false from each disjunct is reported. Although this may be sufficient to allow the flow, it is not always the case. Our suggested strategy would be that PolSim should continue the simulation assuming that a specific predicate evaluated to true and investigate further. This would allow PolSim to give the policy developers more suggestions on which predicates to consider. This further exploration can also be interactive.

Coverage testing parallelisation

Coverage testing can be parallelised as a straightforward next step. Currently a need for that does not arise since each scenario with all input parsing, compilation, simulation, graph and report generation takes less than a second. However, with longer pipelines and with multiple complex policies simulated at once, parallelisation might be needed.

6. REFERENCES

[1] DataLossDB: Open Security Foundation. http://datalossdb.org.

[2] Network Simulator. https://www.nsnam.org.

[3] Privacy Rights Clearinghouse. http://privacyrights.org.

[4] The Network Simulator ns-2: Validation Tests. http://www.isi.edu/nsnam/ns/ns-tests.html.

[5] Thoth policies for data flows in a search engine. https://www.dropbox.com/s/rovt6i70y0npjlf/t3tr.pdf.

[6] Adobe data breach more extensive than previoulsy disclosed. http://www.reuters.com/article/2013/10/29/us-adobe-cyberattack-idUSBRE99S1DJ20131029, Dec. 2013.

[7] Target breach worse than thought, states launch joint probe. http://www.reuters.com/article/2014/01/10/us-target-breach-idUSBREA090L120140110, Jan. 2014.

[8] Comelec Data Breach, 55 million registered Filipino voters vulnerable to identity theft. http://www.aim.ph/blog/5-it-security-lessons-from-the-comelec-data-breach/, Mar. 2016.

[9] Data Breach At Oracle's MICROS Point-of-Sale Division. http://krebsonsecurity.com/2016/08/data-breach-at-oracles-micros-point-of-sale-division/, Aug. 2016.

[10] Leaked: 154 million US voter records expose preferences on gay marriage and abortion law. http://www.ibtimes.co.uk/A6Zes, Jun. 2016.

[11] A. Barth, J. C. Mitchell, A. Datta, and S. Sundaram. Privacy and utility in business processes. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, 2007.

[12] D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV)*, 2010.

[13] M. Y. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, 2007.

[14] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[15] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (S&P)*, 2002.

[16] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[17] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel. Thoth: Comprehensive policy compliance in data retrieval systems. 2016.

[18] R. Fernández and J. Garcıa. Rsim x86: A cost-effective performance simulator.

[19] D. Garg, L. Jia, and A. Datta. Policy auditing over incomplete logs: theory, implementation and applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[20] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[21] K. Hayati and M. Abadi. Language-based enforcement of privacy policies. In *Proceedings of the 4th International Conference on Privacy Enhancing Technologies*, PET'04, pages 302–313, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] D. Hedin and A. Sabelfeld. A perspective on information-flow control.

[23] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[24] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th Symposium on Practical Aspects of Declarative Languages*, 2003.

[25] P. D. Mohamed Alzayat and D. Garg. Polsim: Automatic policy validation via meta-data flow simulation. In *Master Thesis, Saarland University*, 2016.

[26] Y. Mundada, A. Ramachandran, and N. Feamster. Silverline: Preventing data leaks from compromised web applications. In *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[27] A. C. Myers. JFlow: Practical mostly-static information flow control. In *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999.

[28] B. Niu and G. Tan. Efficient user-space information flow control. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013.

[29] F. Ryckbosch, S. Polfliet, and L. Eeckhout. Fast, accurate, and validated full-system software simulation. 2010.

[30] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, Sept. 2006.

[31] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer*

Security, 17(5):517–548, 2009.

[32] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing. Bootstrapping privacy compliance in big data systems. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[33] D. K. Vollmar and D. P. Sanderson. A mips assembly language simulator designed for education. *J. Comput. Sci. Coll.*, 21(1):95–101, Oct. 2005.

[34] Wikipedia. Data breach: Major incidents. http://en.wikipedia.org/wiki/Data_breach#Major_incidents.

[35] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[36] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[37] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System support for derived data management. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2010.

APPENDIX

So far In the use case considered in chapter 4, the conduits IDs were assumed not to be confidential by themselves. Thus we allowed declassifying them without further restriction once `ONLY_CND_IDS` holds. If the conduit IDs themselves are confidential, in the sense that the presence or absence of a conduit ID in the search results may leak sensitive information, we would require that **read** and **declassify** rules of the conduits containing the list of conduit IDs be at least as restrictive as the corresponding rules of all conduits having a conduit ID in the list. For that purpose, Thoth policy developers suggest the macro ONLY_CND_IDS_PLUS (ONLY_CND_IDS+) which as discussed in Chapter 2 expands to:

```
cCurrLenIs(CurrLen) ∧ cNewLenIs(NewLen) ∧
each in(this, CurrLen, NewLen) willsay(CndId)
{cIdExists(CndId) ∧ hasPol(CndId , P) ∧
 isAsRestrictive(read, P.read) ∧
 isAsRestrictive(declassify, P.declassify)}
```

We can simply modify every occurrence of `ONLY_CND_IDS` to become `ONLY_CND_IDS_PLUS`, and the flow will not be blocked by the simulator; the simulator doesn't evaluate the macro in our first example but compares the conditions syntactically as we specified isAsRestrictive(**update**,`ONLY_CND_IDS`). When deploying the modified policy, with `ONLY_CND_IDS_PLUS`, the pipeline will get blocked because the search results do not have **read** and **declassify** rules as restrictive as all conduits whose conduit IDs are mentioned. **That's why we recommend letting PolSim evaluate the macro unless the developer is sure that the macro can be syntactically checked (as for `ONLY_CND_IDS`).** To allow PolSim to expand and evaluate the macro, we modify the **declassify** rule of source files to:

```
declassify :- isAsRestrictive(read, this.read) until
        [[cIsIntrinsic ∧ ONLY_CND_IDS)] ∨
        [Extrinsic ∧ isAsRestrictive(read, this.read)]]
```

This would allow PolSim to evaluate the `ONLY_CND_IDS` macro. Clearly, this would require some changes in the policy; such as giving hints to PolSim about which conduit IDs will be actually written.

This hint can be given to PolSim via the conduit state of the search results conduit.

```
ConduitState:[cCurrLenIs(0),cNewLenIs(3), NewData(
"whiteListedConduitIDs/AliceAccessibleFiles.txt")]
```

This allows loading the data in AliceAccessibleFiles.txt from the file system into the search results conduit within the PolSim environment. Now the search results conduit contains three lines[4];
Alice
AlicesFriends
PublicContent
This modification is enough to have the policies compliant again, and we no more need to have the search results update rule be `ONLY_CND_IDS`.

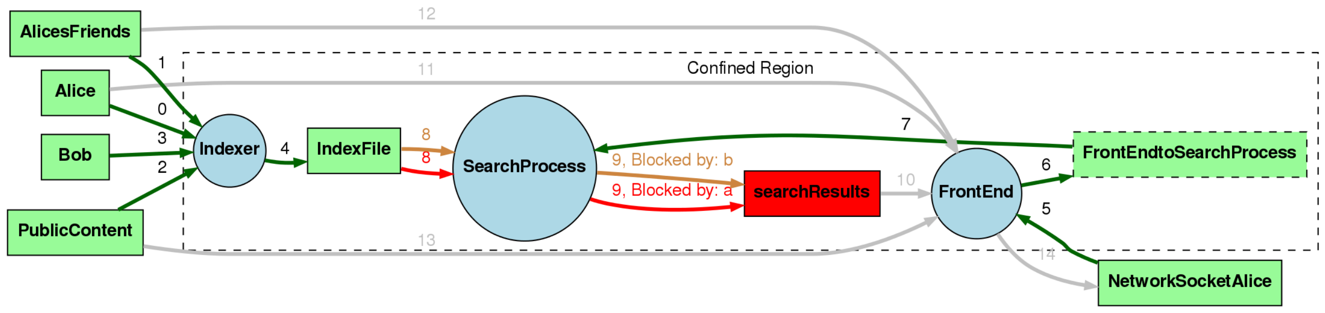
Replacing `ONLY_CND_IDS` with `ONLY_CND_IDS_PLUS`, would result in flow block again as we can see in Figure 9. The reason behind the blockage was that the search results policy didn't meet the requirements of the `ONLY_CND_IDS_PLUS` macro, namely that the **read** and **declassify** rules should be as restrictive as the conduits whose conduit IDs are in the list. The search results conduit should have the **read** rule allowing only the request issuing client to read the data (i.e. Alice in our case) and the **declassify** rule allowing results be declassified to Alice's network socket, which is outside the confined region. A snapshot of the search results conduit policy would be:

```
read :- sKeyIs("Alice")
update :- true
declassify :- isAsRestrictive(read, this.read) until
        [Extrinsic ∧ isAsRestrictive(read, this.read)]]
```

This would suffice to make the simulation not block because the policy installed on the search results would be indeed as restrictive as the original conduits which were outputted by the non-malicious search process. A next step would be to allow censorship. A provider may require that certain documents should be censored when a query comes from some particular country. This can be achieved using the predicates sIpIs(`IP`) and IpPrefix(`IP,Region`). Censorship can be specified in multiple ways. A possible approach is to maintain a set of censored conduits per region. Alternatively, we can maintain a set of censored regions per conduit. Practically, the first approach would be chosen for efficiency reasons. A list of censored conduits is maintained for every region. Both approaches require the modification of all source conduits **declassification** rules to express censorship. For this example, we will model the first approach as it is indeed more efficient for runtime environments with millions of conduits and dynamically changing region IPs, and because it is the approach used in Thoth. All **declassification** rules must be modified to consider the special macro (CENSOR(`cndID`)) which allows censoring their files if required. This macro expands to:

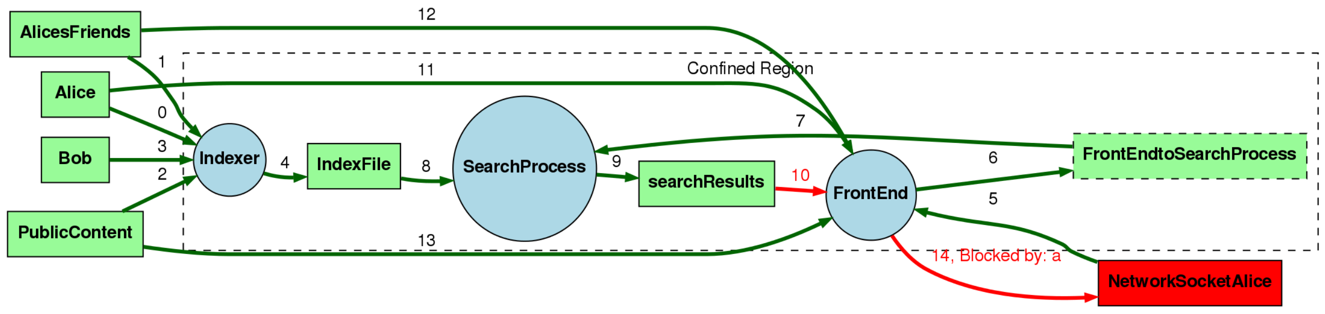
```
sIpIs(IP) ∧ IpPrefix(IP, R)∧
cCurrLenIs(off1) ∧ cNewLenIs(off2)
∧concat(FBL, R, "BlackList")
eachin(FBL, off1, off2) says (c) {neq(c, cndID)}
```

[4]In PolSim we can use alphanumeric ids. For simplicity, conduit names maps to the conduit IDs in a one to one mapping



Blocking predicates	
a	ONLY_CND_IDS_PLUS :-> isAsRestrictive(READ, [[SKeyIs(Alice)]])
b	isAsRestrictive(READ, [[false]])

Figure 9: Search engine simulation with confidential conduit IDs



Blocking predicates	
a	CENSOR :-> Neq(3,3)

Figure 10: Search engine simulation with censorship

The alternative way of modeling it is to censor a conduit from a list of region IPs, which can be done using the special macro CENSOR(blackListFile).

sIpIs(IP) ∧ eachin(blackListFile, off1, off2) says (ipRange) {!IpPrefix(IP, ipRange)}

For simulation purposes, let's assume that the search process is faulty and will write a list of results that include a censored file (the public content file, for instance). The simulation will instantly show that the data flow was blocked at the network socket authenticated with, say an IP from Germany. This will be reported as shown in figure 10.

PolSim reports that the condition neq(3,3) was not satisfied, because neq(3,3) evaluated to false; where 3 is the conduit ID of the public content conduit. In this simulation we instruct PolSim to hide the conduits of the blacklists from the reported graph so as not to have a graph full of blacklists for each region.

Indeed, the policies we arrived at are exactly those presented in the Thoth paper. The description above explains how PolSim can iteratively and interactively help design policies.