Universität des Saarlandes
Max-Planck-Institut für Informatik (IMPRS-CS)
Max Planck Institute for Software Systems
(MPI-SWS)

PolSim: Automatic Policy Validation via Meta-Data Flow Simulation

Technical Report MPI-SWS-2016-005

Masterarbeit im Fach Informatik
Master's Thesis in Computer Science
von / by

Mohamed Alzayat

angefertigt unter der Leitung von / supervised by

Prof. Dr. Peter Druschel
Dr. Deepak Garg

betreut von / advised by

Prof. Dr. Peter Druschel
Dr. Deepak Garg

begutachtet von / reviewers

Prof. Dr. Peter Druschel
Dr. Deepak Garg

Saarbrücken, Tuesday 27th September, 2016

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, Friday 19th January, 2018 Mohamed Alzayat

Abstract

Every year millions of confidential data records are leaked accidentally [7, 6, 1, 3, 8, 10, 9, 34] due to bugs, misconfiguration, or operator error 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, and to enforce relevant declarative policies at processes boundaries, regardless of bugs, misconfiguration, and compromises in application code, or actions by unprivileged operators. However, designing policies that make sure all and only compliant flows are allowed remains a complex and error prone process. In this thesis, 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.

Acknowledgements

In the name of Allah, the Most Gracious and the Most Merciful

First and above all, I praise Allah, the almighty for providing me this opportunity and granting me the capability to proceed successfully. This thesis appears in its current form due to the assistance and guidance of several people. I would, therefore, like to offer my sincere thanks to all of them.

Special appreciation and gratitude go to my supervisors Prof. Dr. Peter Druschel and Dr. Deepak Garg for their supervision and constant support. Their invaluable help by constructive comments and suggestions throughout my thesis work have contributed to the success of this thesis. Their doors were always open whenever I ran into a trouble spot or faced a problem in research, writing, or even had a personal issues. I am blessed having you as supervisors!

I would like to thank Eslam Elnekity for his continuous support throughout the project by offering advice and suggestions on how to approach problems that pop up. I really appreciate your patience teaching me essential research, communication and planning skills. I am extremely thankful for your generosity sharing your valuable expertise and time. Also, I would like to thank Anjo Vahldiek-Oberwagner and Aastha Mehta for their useful suggestions. Special thanks go to Mohamed Gad-Elrab, Akram El-korashy and Omar Darwish for reviewing parts of this thesis and useful suggestions throughout this work.

By the end of this masters program, I am grateful to the International Max Planck Research School for Computer Science, Saarland University, and the Max Planck Institute for software systems family for their continuous support. I am really fortunate being part of this big family. I am also grateful to members of the German University in Cairo for laying down the foundation blocks of my computer science knowledge.

On the personal level, words will never be enough to express how grateful and thankful I am to my parents Wael and Hala, brother Abdelrahman and sisters Safya and Sarah, for their sincere support, encouragement and prayers throughout my life and my long education journey. I really appreciate your patience about my absence, I do miss you! During this work, I was most fortunate to have my family extended by an exceptional member, Lena, my lovely wife. Without your encouragement and prayers, this work wouldn't have been completed. I understand it was and is difficult for you that I am not around most of the time. I needn't say I miss you! Therefore, I can just say thanks for everything and may Allah give you all the best in return.

Moreover, I would like to thank all my Saarbrücken friends. No doubt, I am blessed being surrounded by all those intelligent, caring and enthusiastic personalities. Finally, I would like to extend my sense of gratitude to all my friends and to everyone expressed his support and prayed, or wished the best for me.

Saarbrücken, Germany. Mohamed Alzayat

September, 2016

I dedicate this thesis to my family,
for their constant support and unconditional love.
I love you all dearly!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data retrieval systems like search engines, social networks, online data sharing systems, governmental and healthcare data systems, etc., have sensitive data subject to different policies requiring well engineered systems that guarantee such 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. For example, social content is subject to the policies specified by the content providers, emails are only accessible to their owners, corporate documents are only accessible to employees, and unpublished research papers are only accessible to authors. Search engines have policies regarding the anonymity of queries and privacy of personalization profiles. Moreover, search results must comply with censorship requirements in some countries.

Similarly, Social networks such as Google+ or Facebook, have privacy policies for user posts, likes, photos, etc. These policies can be private, limited to friends, friends of friends or public. Pages and groups also have policies regarding who is allowed to see their posts. Other services such as private messaging have private policies for the parties engaging in conversation.

Ensuring that applicable policies are enforced is a labor-intensive and error-prone process [32]. Enforcing policies requires modifications across multiple layers in the data retrieval system, which makes it hard to reason about and debug policies. Reports of data breaches are increasing, such as [7, 6, 1, 3, 8, 10, 9, 34]. These data breaches cause customers disappointment, company embarrassment, loss of reputation and sometimes lead to fines. This affects business giants as well as startups. When it comes to governments, it's even worse as the leaked data may put 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 developed the Grok system that

combines lightweight static analysis with semi-automatic annotation of source code to enforce many type-specific privacy policies (policies that are applied uniformly to all data of a specific type) in Bing's back-end [32]. Thoth [17] supports individual policies for each data source by having a compliance layer integrated into the Linux kernel, thus can enforce 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 labor-intensive task.

1.1 Motivation and challenges

System policy designers invest large amount of time and effort to develop a working policy that meets all data privacy requirements. Ideally, policy designers should only set policies on data sources and sinks (this would prevent non-compliant flows) and let policy enforcement systems (such as Thoth) ensure policies are correctly enforced. Unfortunately, this is far from reality! To reach a working policy, policy designers have to set the policies for data sources, sinks, and system internal *conduit* (files, pipes, network connections, etc.) which must ensure no compliant flows are denied. Moreover, policy designers need to manually track *flowBlock* (where did the data flow got denied, due to a violated policy) in the data processing pipeline due to violated policies, debug and modify the policies accordingly through an iterative development process till they reach a state where all expected data flows do indeed flow. Even this doesn't guarantee policies are correct nor does it guarantee enough permissiveness to allow all legitimate data flows, as there may be a legitimate flow that was not considered and it may actually get denied during runtime. This is where PolSim comes into play. It tackles the following challenges:

1. The need to think of all possible scenarios that should be allowed given the current system policies and ensuring all legitimate data flows are not blocked
2. The need to manually identify which conduit policy caused the flow to get blocked
3. The need to manually set all intermediate conduit policies

PolSim simulates the policy flow via static analysis of 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 modelling the simulation more intuitive, and still have an 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. Modelling 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)

1.2 Desiderata

From a policy developer perspective, a helpful tool should feature the following:

1. Allow rapid system policies and data flow modelling

2. Give debugging hints that refer to the reason behind policy violation

3. Automatically generate sample policies based on the data flow graph. Policy developers can review the policies afterwards

4. Verify multiple scenarios simultaneously given minimal specifications

5. Report violating data flows in a compact, human readable format

1.3 Outline

The rest of this thesis is structured as follows: In Chapter 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 thesis. In Chapter 3, an overview of PolSim and how it works is given. Chapter 4 shows use cases, 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 Chapter 5.

Chapter 2

Background

2.1 Related work

2.1.1 Data flow simulation

Various kinds of programs operate on data for different goals. 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 as in [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 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 the context of security policy design, simulating data flow within a computer system can help validating a proposed policy. In our work, the development of a tool like PolSim may be helpful for system administrators to ensure security policies do not deny legitimate users of the service –that is being simulated– access to resources. Also, PolSim can helps identify illegitimate access to resources that should be otherwise denied by the security policy under design.

In this sense, PolSim serves as a simulator that aims to help policy designers validate correctness criteria (in this case, criteria like confidentiality and integrity) of a designed security policy.

2.1.2 Information flow control

Confidentiality and integrity are becoming growing concerns [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 (Jif [26] and similar [30]). Jif, for instance, augments Java with information flow types; this can be used in role based [26] 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 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 the abstract labels representing the intended policies. Once this mapping is established, compliance checking can be performed by means of data flow static analysis.

OS kernel-level policy enforcement mechanisms are also widely studied (e.g., Asbestos [16], HiStar [36], Flume [23], Silverline [25], Thoth [17]). There are also other techniques such as software fault isolation (duPro [27]) or hypervisors (Neon [37]) that achieve security policy enforcement goals.

Thoth differs from these systems 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 tracks 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.

Our proposed simulator (PolSim) doesn't enforce policies, and it was never intended to. Instead, it can be used in conjunction with policy enforcement systems (such as Thoth) to make detecting policy violations easier or even help with the policy development process in the first place.

PolSim is fully compatible with the Thoth policy language which 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 [11, 12, 19], respectively. Such languages are known to be clear, concise, and support high-level of abstraction [17]. 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 described in Thoth.

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. PolSim uses static analysis like Grok [32], but for a different purpose, it simulates the flow based on the input policies and the data flow graph, verify the compliance of policies and report the reasons behind policy violations, if any.

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

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 IPC 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 sections we give a summary of what Thoth is, an overview of it’s 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 transactional logs. A global policy store is a fourth *trusted* but centralized policy store 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, no assumptions about the nature of bugs is made. 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". A policy is a full specification of all privacy and integrity rules in effect for the data it gets attached to.

Thoth's policy language can be used to specify the policies of conduits. Access to this 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 more complex example will be discussed later. A condition is expressed as a set of predicates (Table 2.1) connected with conjunction ("and", written $\wedge$) and disjunction ("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 [17]. 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** :- $\mathsf{sKeyIs}(k_{\texttt{Alice}})$
  - **update** :- $\mathsf{sKeyIs}(k_{\texttt{Alice}})$
- **Semi-private data**
  - **read** :- $\mathsf{sKeyIs}(k_{\texttt{Alice}}) \vee (\mathsf{sKeyIs}(K) \wedge (\text{"Alice.acl"}, \texttt{Offset})\ \mathsf{says}\ \mathsf{isFriend}(K))$

  - **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** someone 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. Similarly, this can be extended to support friends of friends privacy requirements.

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. Furthermore, a new predicate can be used, isAsRestrictive(`p1,p2`), 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. 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, 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 that gets expanded only during the policy evaluation (explained in section 2.2.4). Using macros make the policies more readable and easier to understand. The

Relational predicates	
eq(x,y)	x=y
neq(x,y)	x!=y
lt(x,y)	x<y
gt(x,y)	x>y
le(x,y)	x<=y
ge(x,y)	x>=y

arithmetic and string predicates	
add(x,y,z)	x=y+z
sub(x,y,z)	x=y-z
mul(x,y,z)	x=y*z
div(x,y,z)	x=y/z
rem(x,y,z)	x=y mod z
concat(x,y,z)	x = y ‖ z
varType(x,y)	typeof(x) = y

Session predicates	
sKeyIs(x)	x is the session's client authentication key
sIpIs(x)	x is the session's source IP address
IpPrefix(x,y)	x is a prefix of IP address y
timeIs(t)	t is the current time

Conduit predicates	
cNameIs(x)	x is the pathname of conduit
cIdIs(x)	x is the id of conduit
cIdExists(x)	x is a valid conduit id
cCurrLenIs(x)	x is the current conduit length
cNewLenIs(x)	x is the new conduit length
cIsIntrinsic	is this conduit intrinsic?

Content predicates	
(c,off) says	$x_1, \ldots, x_n$ is the tuple
$(x_1, \ldots, x_n)$	found in conduit c at off
(c,off) willsay	ditto for the update
$(x_1, \ldots, x_n)$	of c in the current transaction
each in (c,off1,off2) says	for each tuple in c between off1 and off2, assign
$(x_1, ..., x_n)$ {condition}	to $x_1, \ldots, x_n$ and evaluate condition
each in (c,off1,off2) willsay	ditto for the update
$(x_1, ..., x_n)$ {condition}	of c in the current transaction

Declassification rules	
c1 until c2	condition c1 must hold on the downstream flow until c2 holds
isAsRestrictive(p1,p2)	the permission p1 is at least as restrictive as p2

TABLE 2.1: Thoth policy language predicates and connectives

macro ONLY_CND_IDS expands to

$$\begin{aligned}&\mathsf{cCurrLenIs}(\mathtt{CurrLen}) \wedge \mathsf{cNewLenIs}(\mathtt{NewLen}) \wedge\\ &\mathsf{eachin}(\mathsf{this}, \mathtt{CurrLen}, \mathtt{NewLen})\ \mathsf{says}(\mathtt{Cndid})\\ &\quad \{\mathsf{cidExists}(\mathtt{Cndid})\}\end{aligned}$$

This stipulates that every entry in the newly written data is a valid conduit id. 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:

$$\begin{aligned}&\mathsf{cCurrLenIs}(\mathtt{CurrLen}) \wedge \mathsf{cNewLenIs}(\mathtt{NewLen}) \wedge\\ &\mathsf{eachin}(\mathsf{this}, \mathtt{CurrLen}, \mathtt{NewLen})\ \mathsf{willsay}(\mathtt{CndId})\\ &\{\mathsf{cIdExists}(\mathtt{CndId}) \wedge \mathsf{hasPol}(\mathtt{CndId}\ , \mathtt{P}) \wedge\\ &\ \mathsf{isAsRestrictive}(\mathbf{read}, \mathtt{P}.\mathbf{read}) \wedge\\ &\ \mathsf{isAsRestrictive}(\mathbf{declassify}, \mathtt{P}.\mathbf{declassify})\}\end{aligned}$$

In this macro expansion, $\mathsf{hasPol}(\mathtt{CndId}, \mathtt{P})$ binds P to the policy of the conduit CndId, and the predicates $\mathsf{isAsRestrictive}(\mathbf{read}, \mathtt{P}.\mathbf{read})$ and $\mathsf{isAsRestrictive}(\mathbf{declassify}, \mathtt{P}.\mathbf{declassify})$ enforce 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 access a conduit, the checks made by Thoth depend on whether the process is `CONFINED` or `UNCONFINED`. As shown in the policy enforcement algotithm in Figure 2.1, in `CONFINED` processes, whenever a conduit is being read, its **declassification** rule is added to the reading process taint set. However, when a conduit is being written, all **declassification** rules (c `until` c') must hold in one 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 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 prior; e.g., b is false.

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 2.1: Thoth policy enforcement algorithm

- 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(y, x).

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 that accepts only a list of file names **and** having a police isAsRestrictive(**declassify**, `P`) partial evaluation will allow such policy to propagate if the data is to be written to a file that accepts only a list of file names and having a policy that implies isAsRestrictive(**declassify**, `P`)

Chapter 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.

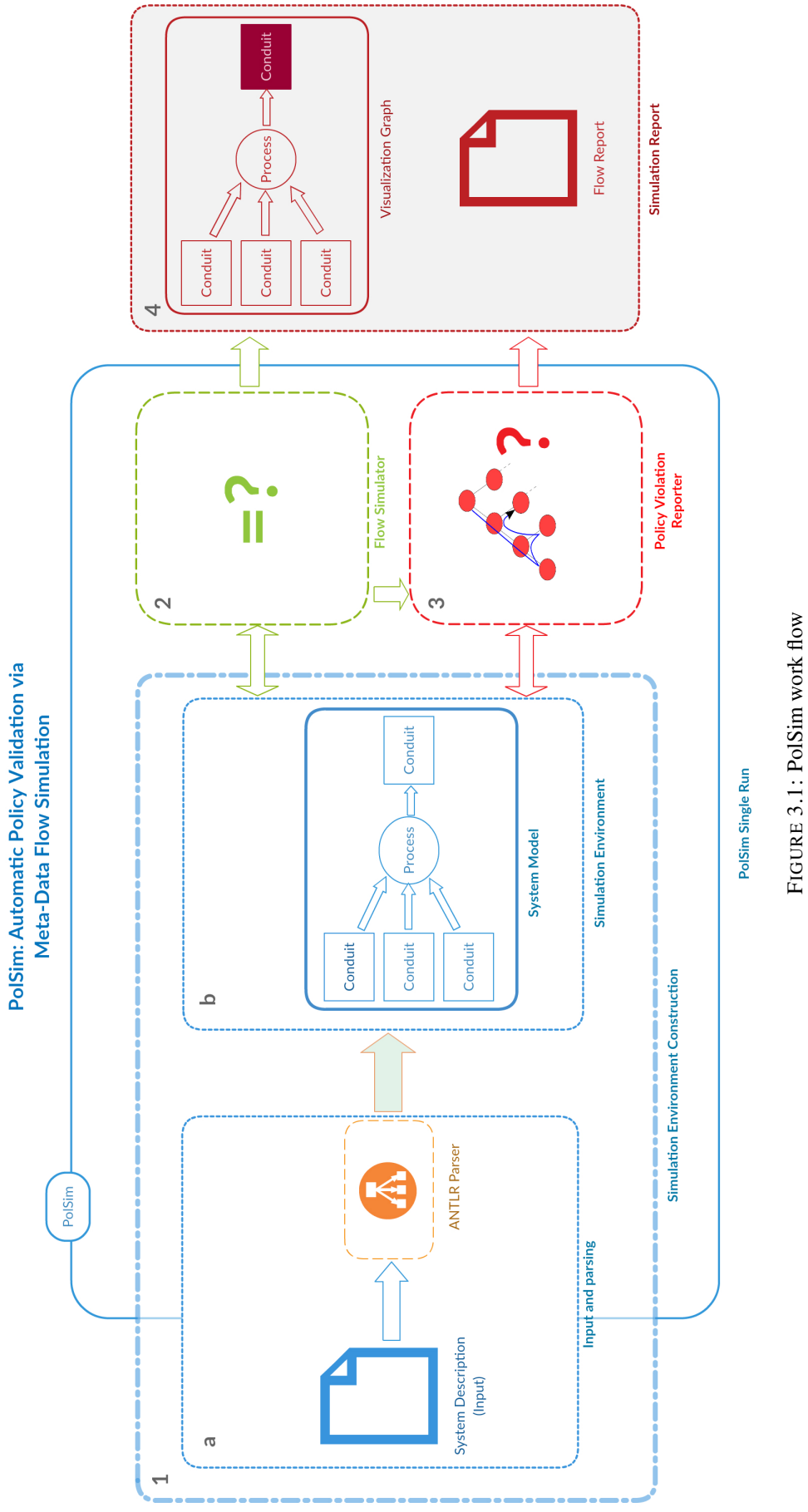


FIGURE 3.1: PolSim work flow

3.1 System overview

As shown in Figure 3.1, PolSim work flow is a four step sequential process:

1. Simulation environment construction:
   (a) A system description file is parsed via an *ANTLR*[28] parser, 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. If a flow gets blocked (denied), the policy violation reporter investigates the causes of the flow block
4. Information from the flow simulator and the violation reporter is used to construct the simulation report

In the following 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 has all policies and configurations needed for the simulation. The system description file should be written by the policy development team in consultation with the system developers and code testers. It follows a specific grammar that uses the Thoth policy language together with few annotations and hints to assist PolSim approximate the dynamic system behavior.
A system description file must include:

- Nodes description
- Flow description

The system description file can optionally include:

- Macros definitions
- User defined (Generic) predicates
- System state definition

The rest of this subsection is dedicated to explaining the inputs that are written in the system description file. For the purposes of explaining the inputs, we consider a **simplified**[1] example of an indexing and search pipeline which was used in Thoth [17].

Nodes description

In PolSim, nodes represent system components, namely conduits and processes (tasks). Conduits represent files, pipes, and network connections. Processes represent tasks running in the system. Conduits' policy-relevant content can be optionally provided as hints, in what we refer to as "conduit state". A conduit state provides enough context for the simulation to better approximate the dynamic system behavior.
A node can be either a conduit or a process. In PolSim input nodes have the following attributes:

1. Conduit
   (a) **Conduit Name:** A conduit identifier that can be used to identify or reference a conduit in policy specifications
   (b) **Conduit Policy:** A single point for specifying all privacy rules relevant to this conduit's data (the data itself is not needed)
   (c) **Conduit State:** Conduit state is where a policy designer abstracts run time data for PolSim to allow simulations with policies that depend on runtime content. In a conduit state a session owner, IP address, time, or some parts of the conduit's runtime content can be specified. This contributes to solving PolSim's first challenge as discussed in Section 1.1.
2. Process
   (a) **Process Name:** A process identifier, helps identify the process during the simulation

So to start with our example: we define two conduits (searchable files), each of them will have a policy, a process that indexes them and writes a third conduit, an index file. Our two conduits would be Alice, and Bob each of them would have a policy that allows reading and updating the file only by its owner with no other declassification rules. So the conduit description

[1] Not a full pipeline, just the indexing part

would be:

$$
\begin{aligned}
&Conduit\ [Alice, Policy : [\\
&\quad Read:\ [[SKeyIs(\text{“}Alice\text{”})]],\\
&\quad Update:\ [[SKeyIs(\text{“}Alice\text{”})]],\\
&\quad Declassify:\ [[[[isAsRestrictive(read, this.read)]]\ UNTIL\ [[false]]]]\\
&\quad];\\
&ConduitState:\ [null]\\
&]
\end{aligned}
$$

Bob's conduit definition would be identical to Alice's except for the name of the conduit, and the key specified.
We define a process Indexer simply by writing:

$$Process[Indexer]$$

Finally we specify the Index file conduit with no custom policy, with a default policy just for now, as follows:

$$
\begin{aligned}
&Conduit\ [IndexFile, Policy : [\\
&\quad Read:\ [[true]],\\
&\quad Update:\ [[true]],\\
&\quad Declassify:\ [[[[isAsRestrictive(READ, this.READ)]]\ UNTIL\ [[false]]]]\\
&\quad];\\
&ConduitState:\ [null]\\
&]
\end{aligned}
$$

Flow description

Flow description is what describes the data flow between conduits and processes. It abstracts the actual data flow and Inter-Process Communication (IPC). It can be specified by the policy designer by specifying ordered edges (representing data flow) between nodes (conduits and processes). Building on our simplified example, we can specify it in PolSim as simple as follows:

$$
\begin{aligned}
&Alice -> Indexer\\
&Bob -> Indexer\\
&Indexer -> IndexFile
\end{aligned}
$$

This acts as a description of how the data flows between system nodes, and how the flows are partially ordered. In practice, one way to generate the data flow graph is to record actual IPCs

and file reads and writes in a production system before policies have been added. This is briefly discussed in Chapter 5.

User defined predicates and their relations

The Thoth policy language features a set of predicates (Table 2.1) that are used in declaring data policies. To allow for further policy abstraction, we introduce user defined predicates which allow policy developers to specify generic predicates with a predefined number of arguments. This gives the designer the flexibility to chose a representative name for what a predicate or a macro may be doing in an actual system. It also allows the policy designer to abstract some of the macros as predicates. This comes handy when the macro expansion is irrelevant or not the focus of the current flows being simulated (e.g., the policy designer assumes the macro is correct and it's expansion will evaluate to true; thus can be abstracted and compared as a unit without expanding and evaluating it).

Consider a rule that allows Alice and her friends to read a file:
read :- sKeyIs($k_{\texttt{Alice}}$) $\vee$ (sKeyIs(K) $\wedge$ ("Alice.acl", `Offset`) says isFriend(K))
For a system that has many semi-private policies, a macro can be introduced for that purpose, thus a FriendsOf(x) macro can be defined to expand to:
sKeyIs($k_{\texttt{x}}$) $\vee$ (sKeyIs(K) $\wedge$ concat(`L, x,".acl"`) $\wedge$ (`L, Offset`) says isFriend(K))
Thus, we can rewrite the simplified read condition as:
read :- FriendsOf(`"Alice"`)

For the purposes of our simulation, expanding the macro during the evaluation may be irrelevant because we are only interested in knowing that this file is only accessible to Alice and her friends. This is where the addition of user defined predicates and their relations becomes useful.

For instance, we can define generic predicates named SKeyIs[1], FriendsOf[1] and FriendsOfFriendsOf[1]. This specification in its own is only useful if predicates will be compared syntactically (predicates hashes comparison). To give an added value for such generic predicates, we introduce a restrictiveness operator "<<" which means that the predicate on the left hand side with a given argument set is at least as restrictive as the predicate on the right hand side given a possibly different argument set. For instance we can specify that SKeyIs is at least as restrictive as FriendsOf which is at least as restrictive as FriendsOfFriendsOf given the same argument by specifying: SKeyIs(x) << FriendsOf(x) << FriendsOfFriendsOf(x). Thus SKeyIs(`Alice`) is at least as restrictive as FriendsOf(`Alice`), but can't be compared to FriendsOf(`Bob`). If this relation is written, there is no need to expand the macro FriendsOf(x)for the purpose of t

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

Thus, for the purposes of our simulation, we can abstract the macro FriendsOf(x), that expands to
sKeyIs(k_x) $\vee$ [sKeyIs(K) $\wedge$ concat(L, x, ".acl") $\wedge$ (L, Offset) says isFriend(K)]
into a very simple user defined predicate FriendsOf(x) provided that we have the condition:
SKeyIs(x) << FriendsOf(x).
In the scope of PolSim, this gives enough information and abstraction to have the data flow analysis possible and yield concise information and debugging hints.

In our running example, we can specify that SKeyIs(x) is at least as restrictive as FriendsOf(x) by simply specifying in the system description file:

$$SKeyIs(x) << FriendsOf(x)$$

we can specify another conduit AlicesFriends using the described predicate:

$Conduit\ [AlicesFriends, Policy : [$
$\quad Read:\ [[FriendsOf(\text{“}Alice\text{”})]],$
$\quad Update:\ [[SKeyIs(\text{“}Alice\text{”})]],$
$\quad Declassify:\ [[[[isAsRestrictive(READ, this.READ)]]\ UNTIL\ [[false]]]]$
$\quad];$
$ConduitState:\ [null)]$
$]$

This contributes to solving challenge number two as discussed in Section 1.1.

System state definition

Similar to the conduit state which allows abstracting runtime data, system state does the same thing for the whole simulation environment. The system state defines states that apply for all conduits, eg: time, specifying time to be t is equivalent to specifying time to be t in every conduit state. This contributes to solving the first challenge listed in Section 1.1. So for our running example, after specifying all conduits and processes, we can specify the system state as follows:

$$SystemState : [timeIs(13/09/2016\ 09:00:00)]$$

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

Policy enforcement is achieved by simulating each and every possible flow of data (subject to the flow description) and applying the checks summarized in the algorithm of Figure 3.2. The enforcement process starts by simulating the flow of policies while respecting the total/partial order of flows input by the policy designer as explained in Section 3.1.1. As in Thoth, declassification rules (as introduced in the policy language overview in Chapter 2) 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 doesn't deal with the runtime environment, we don't have explicit reads or updates. Read and update rules are checked only if referred to during declassification rules.

In order to make the terminology clear we would like to clarify two important terms: a ***compliantFlow*** is a flow that is compliant with the ingress and egress policies, i.e. a flow that doesn't cause data leakage.
a ***flowBlock*** is what happens when a policy violation is detected, whether it was expected or not.

The rest of this subsection is dedicated to explaining the algorithm that PolSim implements to simulate the Thoth policy enforcement mechanisms, and how intermediate conduits policies can be automatically generated.

PolSim policy enforcement algorithm

Figure 3.2 summarizes the abstract checks that PolSim makes when trying to simulate a flow from a conduit to a process or vice versa. If the operation is a read, then PolSim adds the

Question: Should a conduit read or write cause a policy violation?
Inputs: p, the process reading or writing the conduit
f, the conduit(file) being read or written
op, the operation being performed (read or write)
Output: Allow or deny the data flow
Side-effects: May update the taint set of p

```
1  if op is read:
2    Add f's declassification rule to p's taint set.
3  if op is write:
4    // Enforce declassification policies of p's taint set
5    for each declassification rule (c until c') in p's taint set:
6      Check that EITHER c' holds OR (c holds AND
       f's declassification policy implies (c until c')).
```

FIGURE 3.2: PolSim policy validation algorithm

declassification rule of the conduit being read to the taint set of the reading process. No policy check is performed in this case. In order to make sure the taint set doesn't grow un boundedly, we reduce the taint set size by doing taint compression, i.e a policy is not added if the taint set already includes an equally or more restrictive policy.

If the operation is a write, i.e a process is writing to a conduit, then all declassification rules (c `until` c') must hold in one of two ways: either c' holds (thus data can be declassified), or c holds and the conduit being written to enforces (c `until` c').

In our simulation we try declassifying data as early as possible 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 (conjunction of all policies in the taintset). 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!

In our running (simplified) example, the index file[2] would get a policy suggestion that would be as restrictive as all policies in the taint set of the Indexer process. the suggested policy would

[2] the index file is actually an intermediate conduit as it doesn't cross the confinement boundary

be:

$$
\begin{aligned}
&Conduit\ [IndexFile, Policy : [\\
&\quad Read :\ [[SKeyIs(Alice)\&SKeyIs(Bob)]],\\
&\quad Update :\ [[true]],\\
&\quad Declassify :\ [[[[[isAsRestrictive(READ, [[SKeyIs(Alice)]])]]UNTIL[[false]]]\&\\
&\qquad\qquad [[[isAsRestrictive(READ, [[SKeyIs(Bob)]])]]UNTIL[[false]]]]]\\
&\quad];\\
&ConduitState :\ [null]\\
&]
\end{aligned}
$$

Although this policy can never be satisfied, it actually gives a hint that this file's policy should allow no one to read it (which is the expected policy, since the index consumes data private to both Alice and Bob).

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 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, and thus was propagated through the data processing pipeline).
2. The list if 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 heuristic, 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, 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.

In our running example, if the index file had a non default policy, and assuming we modify the **declassify** rule to always allow declassifying (**declassify** :- `true until true`), this would cause the flow to be denied because the declassification rules of the source conduits cannot be enforced. PolSim would report the path from Alice's conduit to the index file conduit passing through the Indexer process as seen in Figure 3.3.

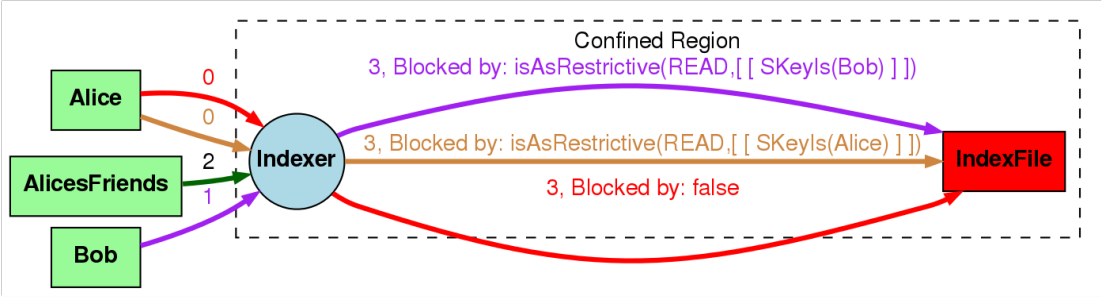


FIGURE 3.3: Simplified search engine violation reporting

In Figure 3.3 we can see Green rectangles that represent conduits, pale blue circles represent processes, sequenced directed edges show the sequence and flow of data in the modeled system. The conduit where the flow got blocked at is colored in red, denied flows are colored with colors other than green.

Declassification rules of the ingress policies are in the form (c1 `until` c2), PolSim reported the blocking predicate false as the c2 of Alice's conduit's policy part was not satisfied. PolSim also reported isAsRestrictive(**read**, [[SKeyIs(`Alice`)]]) and isAsRestrictive(**read**, [[SKeyIs(`Bob`)]]) because the c1 part wasn't satisfied (evaluated to false) when trying to propagate the policies forward.

This information is enough to debug the policy, namely, to realize that either the IndexFile's declassification rule must be more restrictive **or** both Alice's and Bob's source files must have more permissive declassification policies. Since PolSim doesn't know the intent of the policy writer, it cannot say which of these two is the correct solution.

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 (*conduits in lime green, processes in pale blue*)
   (c) Explored paths (*explored paths in green, unexplored paths in gray*)
   (d) Conduits with automatically generated policies (*lime green with dashed borders*)
   (e) The conduit where the flow got blocked (*red*)
   (f) Blocking macros and predicates (*different colors: red, brown, purple, ...*)
   (g) A trace of all blocking predicates paths (*same color as the blocking predicate/macro text*)
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

This contributes to solving the fourth challenge discussed in Section 1.1. An example of the simulation reports of this simplified example and slightly more complex ones can be found in the appendix.

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 2.1) 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 doesn't 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).

Chapter 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 boost and make developing a policy that allows all compliant flows much easier. As an input to PolSim we need the following:

- Conduits and Processes
- Conduit source 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:

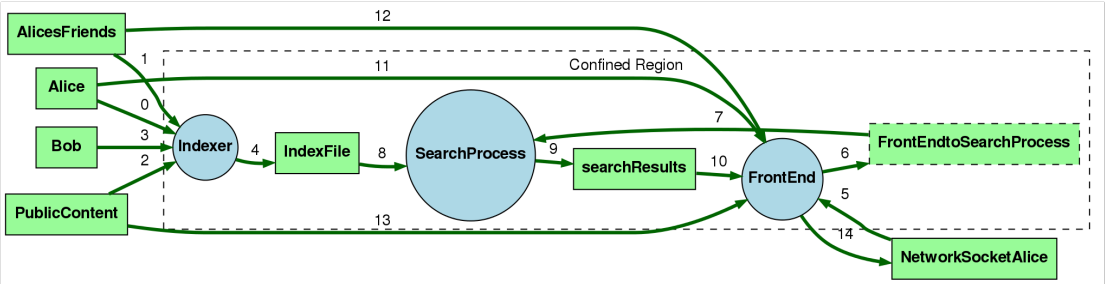


FIGURE 4.1: Search engine data processing pipeline[1]

The search pipeline as shown in Figure 4.1 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)

[1] as generated by PolSim

- **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):

declassify :- isAsRestrictive(**read**, this.**read**) until false

For the simulation, 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 4.2:

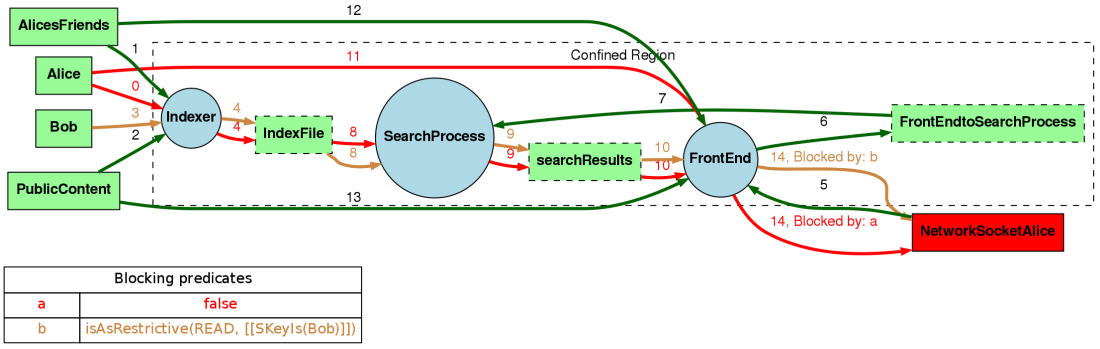


FIGURE 4.2: First step towards developing a search engine policy[2]

Flow blocking predicate: isAsRestrictive(READ,[[SKeyIs(Bob)]])
Flow blocking conduit: NetworkSocketAlice
Flow blocking predicate origin: Bob
Flow blocking Permission: [[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[false]]] ∧ [[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[false]]]]]]

Flow blocking predicate: false
Flow blocking conduit: NetworkSocketAlice
Flow blocking predicate origin: Alice

[2]graph optimized for printing by referring to the blocking predicates outside the graph

Flow blocking Permission: [[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[false]]] ∧ [[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[false]]]]]]

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 blocked! 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 taint set have 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:

$$\begin{aligned}&\mathsf{cCurrLenIs}(\mathtt{CurrLen}) \wedge \mathsf{cNewLenIs}(\mathtt{NewLen}) \wedge \\ &\mathsf{each\ in}(\mathsf{this}, \mathtt{CurrLen}, \mathtt{NewLen})\ \mathsf{says}(\mathtt{Cndid}) \\ &\quad \{\mathsf{cidExists}(\mathtt{Cndid})\}\end{aligned}$$

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:

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

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 macro will happen when actual data starts to be written to the conduit.
Alternatively, the policy can be written

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

This case requires expanding and evaluating the macro whenever there is a potential declassification (i.e. a process is checking if it can declassify to a conduit).

PolSim can expand and evaluate macros, but to keep the first example simple, we will go for the first method which allows declassifying iff the update rule of the conduit, which data would be declassified to, is at least as restrictive as `ONLY_CND_IDS`. We set the first declassify rule above on the IndexFile.

We run the simulation again, as we can see in Figure 4.3, there is still a problem.

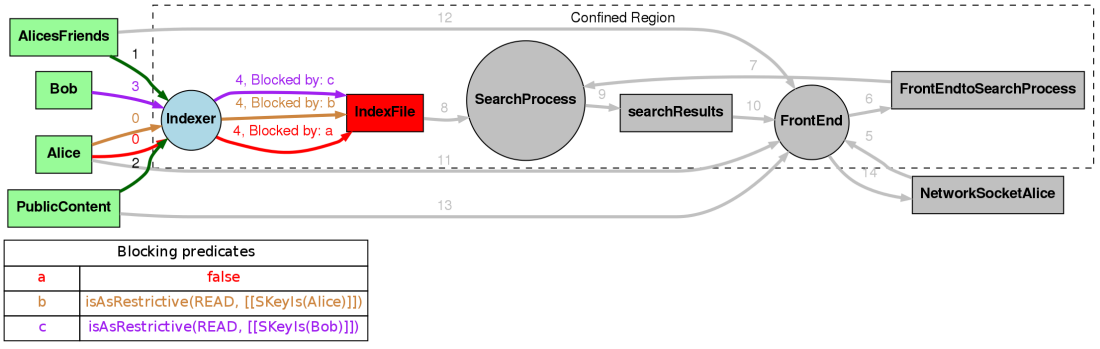


FIGURE 4.3: Search engine simulation with an index policy

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:

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

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

However, modifying the source files still results in the pipeline stalling again as we can see in Figure 4.4:

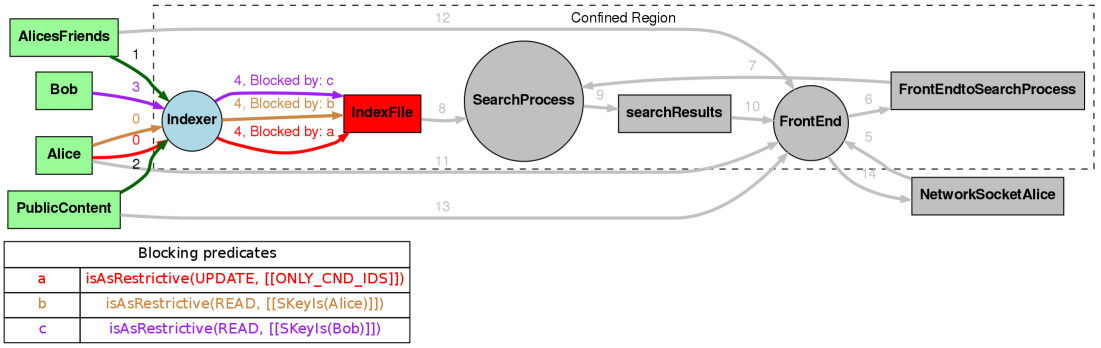


FIGURE 4.4: Search engine simulation with modified ingress policies

Now the flow blocking predicates are: isAsRestrictive(**update**, `ONLY_CND_IDS`), isAsRestrictive(**read**, [[SKeyIs(`Alice`)]]) and isAsRestrictive(**read**, [[SKeyIs(`Bob`)]]). The issue now is that the index file's read policy is still permissive, while the source declassification policies forbid this. The solution is to change the IndexFile's read policy to false:

read :- 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 4.5, 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

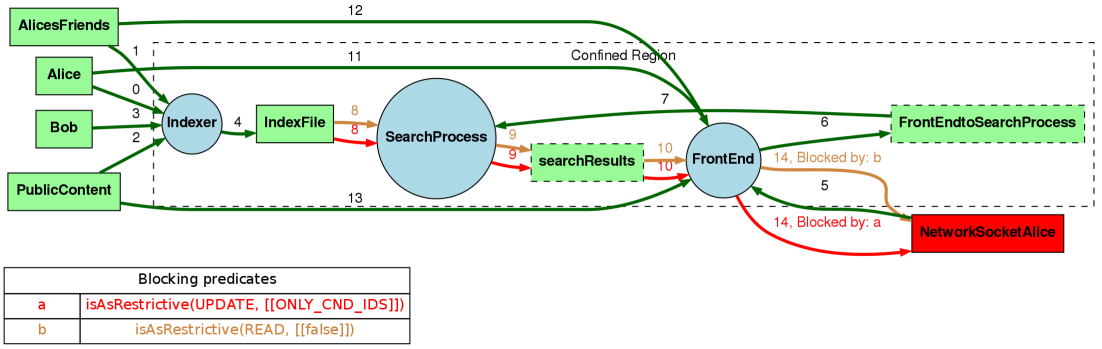


FIGURE 4.5: Search engine simulation with modified ingress and index file policies

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`.

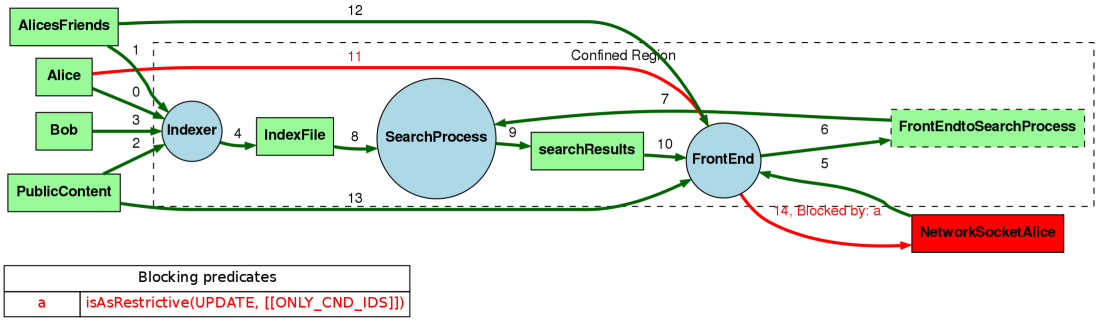


FIGURE 4.6: Search engine simulation with modified search results policy

This modification resulted in a flow block, the flow blocked due to the fact that the network socket doesn't 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:

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

[3]Extrinsic is equivalent to not intrinsic

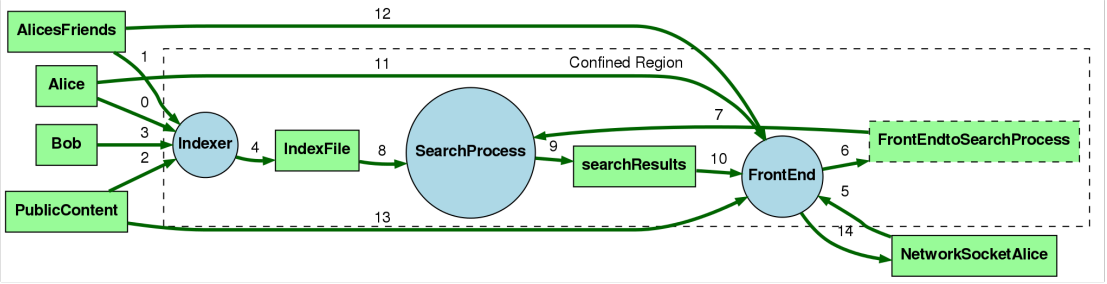


FIGURE 4.7: Minimal search engine data processing pipeline

This would result in the simplest search pipeline that respects the provider's policies. The simulation would indeed succeed as in Figure 4.7.

So far, 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:

$$\begin{aligned}
&\mathsf{cCurrLenIs}(\texttt{CurrLen}) \wedge \mathsf{cNewLenIs}(\texttt{NewLen}) \wedge \\
&\mathsf{each\ in}(\mathsf{this}, \texttt{CurrLen}, \texttt{NewLen})\ \mathsf{willsay}(\texttt{CndId}) \\
&\{\mathsf{cIdExists}(\texttt{CndId}) \wedge \mathsf{hasPol}(\texttt{CndId}\,, \texttt{P}) \wedge \\
&\ \mathsf{isAsRestrictive}(\textbf{read}, \texttt{P}.\textbf{read}) \wedge \\
&\ \mathsf{isAsRestrictive}(\textbf{declassify}, \texttt{P}.\textbf{declassify})\}
\end{aligned}$$

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:

$$\begin{aligned}
&\textbf{declassify}\ \text{:-}\ \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read})\ \texttt{until} \\
&\qquad [[\mathsf{clsIntrinsic}\ \wedge\ \texttt{ONLY_CND_IDS})]\ \vee \\
&\qquad [\mathsf{Extrinsic} \wedge \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read})]]
\end{aligned}$$

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 4.8.

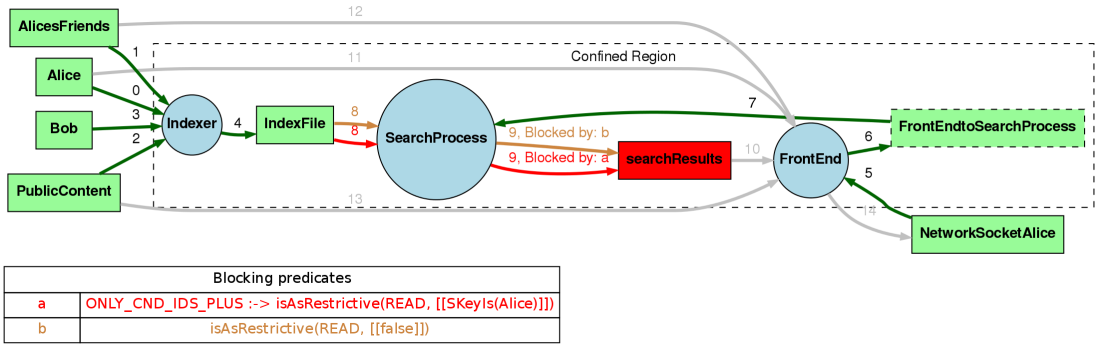


FIGURE 4.8: Search engine simulation with confidential conduit IDs

The reason behind the blockage was that the search results police 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:

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

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, as we can see in Figure 4.9.

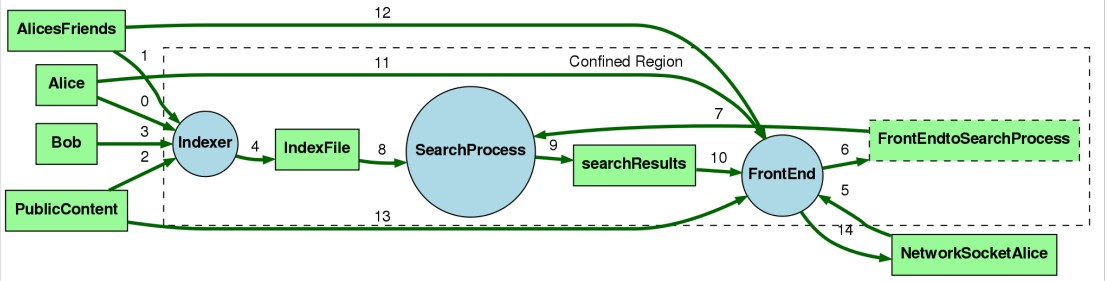


FIGURE 4.9: Search engine data processing pipeline

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)}

The alternative way of modelling 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 4.10

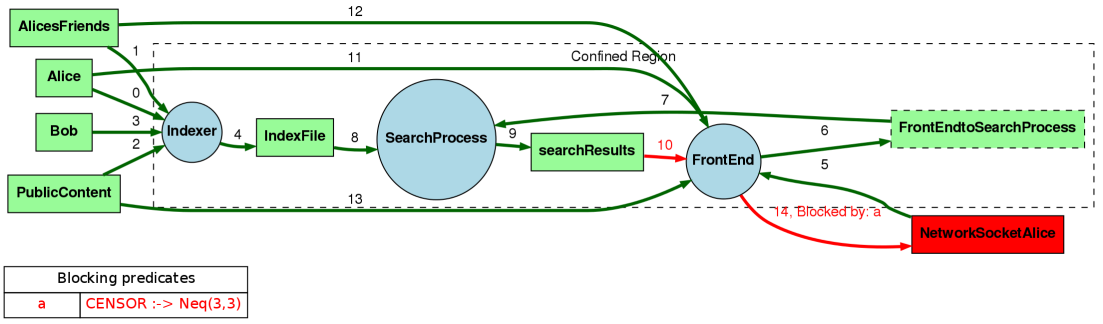


FIGURE 4.10: Search engine simulation with censorship

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.

We also modeled the personalization and advertisement pipeline from the Thoth paper and found no problems, discussed briefly in the appendix.

4.1.2 Evaluation

In this Chapter, 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 modelling 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.

This contributes to solving challenge number three from Section 1.1. An example of PolSim's coverage testing output can be seen in the Appendix.

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). However, having thousands of scenarios will take more time. Simulation time depends on many factors such as when the simulation stalls in the pipeline, whether will the restrictiveness checks 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 should scale well.

Chapter 5

Scope and Outlook

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.

5.1 Scope

PolSim is a testing tool, it's coverage is not complete. PolSim verifies the compliance of the source and sink policies by simulating their flow and simulating their enforcement. PolSim approximates how the flow will behave in a deployed system given the set of policies provided as an input. Unfortunately, we can not guarantee that a set of policies that appear to allow all expected flows in PolSim will allow all expected flows in a deployed system, because there may be some runtime-data sensitive predicates that would cause a violation when executed on runtime-data rather than the sample data provided by the developer.

5.2 Outlook

5.2.1 Automatic simulation environment generation

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.

5.2.2 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.

5.2.3 Input range coverage testing

Having multiple scenarios simulated and compared together is a 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.

5.2.4 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.

5.2.5 Coverage testing parallelisation

Coverage testing can be parallelised in a straight forward manner. Currently a need for that doesn't 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.

Appendices

Appendix A

Personalization and advertisement pipelines

Augmenting the search pipeline with personalization and advertisement pipelines (as discussed in Thoth) is a typical extension that most data retrieval systems have. We show how to develop policies to allow for such personalization and targeted advertisements components while maintaining a policy that would allow all compliant flows.

The new pipeline will looks as follows:

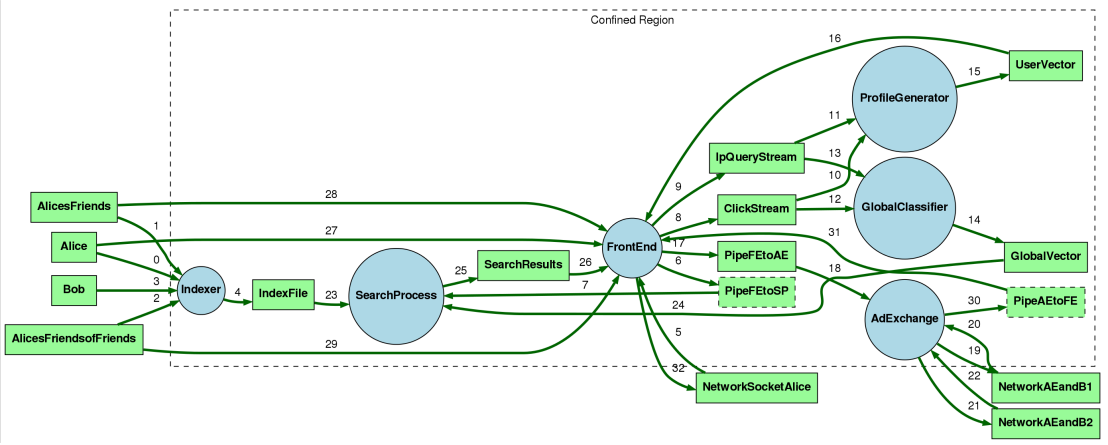


FIGURE A.1: Search engine, personalization and advertisement data processing pipeline[1]

- Indexer Process indexes all searchable files in the system
- Indexer writes the index file for later lookups

[1] as generated by PolSim

- The Profile generator reads the the IP and query history (IpQueryStream) and the click history (clickStream) and updates the user's profile vector
- Similarly the global classifier reads the IpQueryStream and clickStream and updates the system global preference vector
- End user (Alice) authenticated with a session key $\boldsymbol{K}$ connect to the network socket
- The end user issues a query to the front end
- The front end forwards the query to the ad exchange together with information from the search and clicks history (fetched from the user vector)
- Ad exchange process connects to multiple brokers for bids
- Ad exchange collects all bids and forwards the winning broker ad campaign to the front end
- In the mean time, the front end forwards the query to the search process
- The search process fetches the index and the global vector to select search results
- 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, re-order the search results based on the user vector, insert relevant advertisements and writes back the query results to the network socket

Personalizing the search results for a client requires maintaining some information about the user. Typically information such as search history, click history and profile details are maintained for such personalization. The front end uses this information to enhance search queries, re-order the search results and select ads for inclusion in the results.

We would like to highlight the similarities between the personalization, advertisements and the search pipeline we have discussed. In the search pipeline, we used to declassify the conduits IDs. Similarly, for the personalization pipeline we would declassify a vector of floats representing the user preferences. In the advertisements pipeline, we will declassify the search query, and the user preference vector.

The personalization pipeline depends on the existence of four main conduits and two main processes. The conduits are a query stream, click stream, global vector and user vector. The global vector stores data generated from the query and click streams of all users. User vector file stores user personalization information including search and click histories. Moreover, the query and click streams can be consumed only within 2 days. The processes are a profile generator and a global classifier that consumes periodically the query and click streams to regenerate the

user vector and the global vector respectively. The query and click stream conduits share the following policy:

For that we need to set policies for the query and click streams, global vector and user vector. The global vector consumes data from all users and is consulted by the search process, thus should get a policy identical to the index file:

- **read** :- false
- **update** :- true
- **declassify** :- isAsRestrictive(**read**, `this`.**read**) `until` clsIntrinsic $\wedge$ `ONLY_CND_IDS_PLUS`

The query stream as well as the click stream will get the same policy; a policy that would allow the data be consumed within 48 hours and can be transformed only to a user preference vector of floats. Moreover, the data derived from this vector can only be declassified to list of conduit IDs or to an extrinsic conduit authenticated by the user. Thus their policies will be:

- **read** :- SKeyIs(`Alice`)
- **update** :- true
- **declassify** :- [isAsRestrictive(**read**, `this`.**read**) `until` [isAsRestrictive(**update**, `ONLY_FLOATS`) $\wedge$ clsIntrinsic $\wedge$ collectedTime(`ct`) $\wedge$ currentTime(`tnow`) $\wedge$ sub(`diff`, `tnow`, `ct`) $\wedge$ lt(`diff`, 172800)]] $\wedge$ [isAsRestrictive(**read**, `this`.**read**) `until` [[clsIntrinsic $\wedge$ `ONLY_CND_IDS_PLUS`] $\vee$ [Extrinsic $\wedge$ isAsRestrictive(**read**, `this`.**read**)]]]

We use the two generic predicates collectedTime and currentTime to allow binding the variables with the runtime hints we provide PolSim with.

Since PolSim supports partial evaluation as Thoth does. The user vector should satisfy the first conjunct by having an update rule that doesn't accept except float vectors `ONLY_FLOATS`. The read rule should allow only the owner of the data used to generate this user vector (i.e. Alice). Finally, the user vector's **declassification** rule should be as restrictive as the second conjunct of the query and click streams' declassification `until` part:
[isAsRestrictive(**read**, `this`.**read**) `until` [clsIntrinsic $\wedge$ `ONLY_CND_IDS_PLUS`] $\vee$ [Extrinsic $\wedge$ isAsRestrictive(**read**, `this`.**read**)]]

The extended pipeline as generated by PolSim will look as follows:

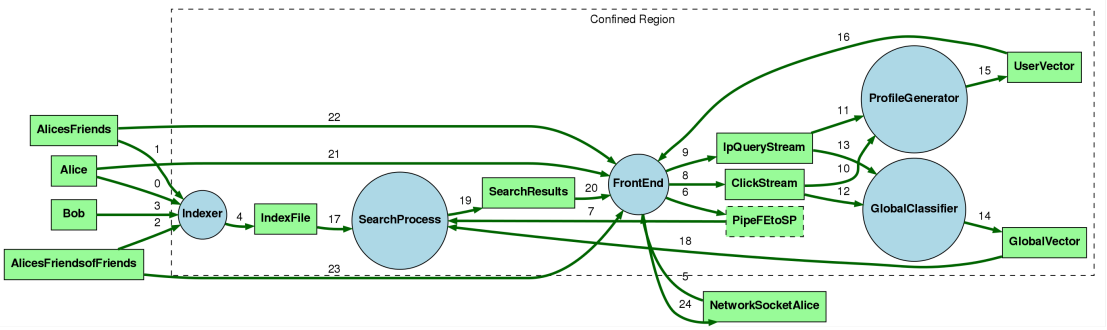


FIGURE A.2: Search and indexing pipeline with personalization

The advertisement pipeline consists of an ad exchange process that sends the search query and the user personalization vector to the advertisement bidders for getting bids, the winning campaign gets his advertisements IDs forwarded to the front end for insertion in the results page that will be sent to the client. This requires the declassification of the search query and the personalization vector (declassifying the search query and a vector of floats) to the network sockets used for communicating with the bidders.

The full pipeline is now constructed and the policies allow all compliant flows given the system description provided. See Figure A.3

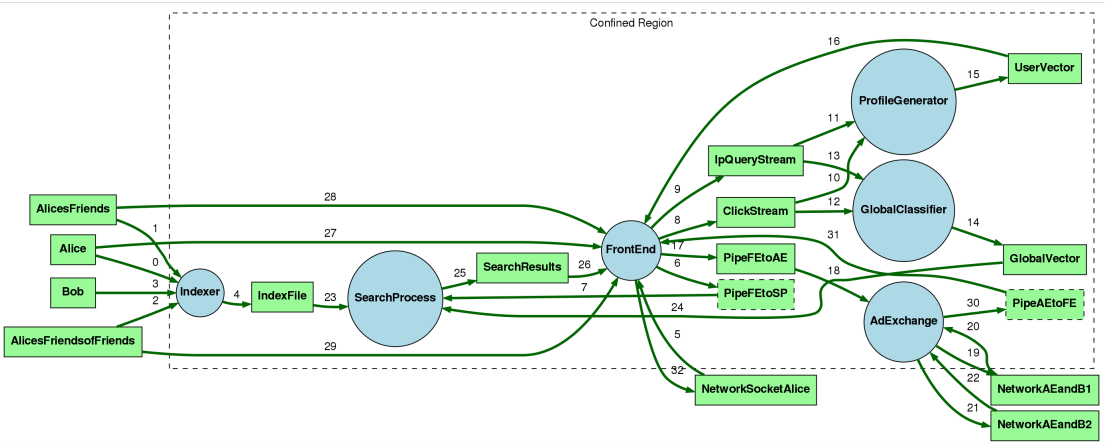


FIGURE A.3: Search and indexing pipeline with personalization and advertisements

Appendix B

Simplified search engine pipeline report

PolSim SUMMARY GRAPHKEY DECLASSIFICATIONS BLOCKERS

Policy Simulation Summary

THIS RUN WAS BLOCKED!

Flow BLOCKING PREDICATES [false, isAsRestrictive(READ,[[SKeyIs(Alice)]]), isAsRestrictive(READ,[[SKeyIs(Bob)]])]
++++++++++++++++++++++++++++

{Flow blocking predicate: false }
Flow blocking conduit: IndexFile
Flow blocking predicate origin: Alice
Flow blocking Permission: [[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[false]]] & [[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[false]]]]]]
++++++++++++++++++++++++++++

{Flow blocking predicate: isAsRestrictive(READ,[[SKeyIs(Alice)]]) }
Flow blocking conduit: IndexFile
Flow blocking predicate origin: Alice
Flow blocking Permission: [[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[false]]] & [[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[false]]]]]]
++++++++++++++++++++++++++++

{Flow blocking predicate: isAsRestrictive(READ,[[SKeyIs(Bob)]]) }
Flow blocking conduit: IndexFile
Flow blocking predicate origin: Bob
Flow blocking Permission: [[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[false]]] & [[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[false]]]]]]
++++++++++++++++++++++++++++

Blocked!
END OF RUN#0_0

GRAPH KEY?

GRAPH KEY

Alice
EConduit [Alice,
Policy: [
Read: [[SKeyIs(Alice)]],
Update: [[SKeyIs(Alice)]],
Declassify: [[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[false]]]]]]
]

ConduitState: [SKeyIs : Alice]]

AlicesFriends
EConduit [AlicesFriends,

FIGURE B.1: Simplified search engine pipeline report part 1

Policy: [
Read: [[FriendsOf(Alice)]],
Update: [[SKeyIs(Alice)]],
Declassify: [[[[[[isAsRestrictive(READ,[[FriendsOf(Alice)]])]] UNTIL [[false]]]]]]
]

ConduitState: [SKeyIs : Alice]]

Bob

EConduit [Bob,
Policy: [
Read: [[SKeyIs(Bob)]],
Update: [[SKeyIs(Bob)]],
Declassify: [[[[[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[false]]]]]]
]

ConduitState: [SKeyIs : Bob]]

Indexer

Process [Indexer, policies=
[[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[false]]] & [[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[false]]]]]]]

IndexFile

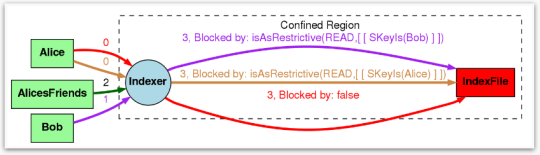
Conduit [IndexFile,
Policy: [
Read: [[true]],
Update: [[true]],
Declassify: [[[[[[isAsRestrictive(READ,[[true]])]] UNTIL [[false]]]]]]
]

ConduitState: []]

SUCCESSFUL DECLASSIFICATIONS

Declasifications

There were no successful declassifications

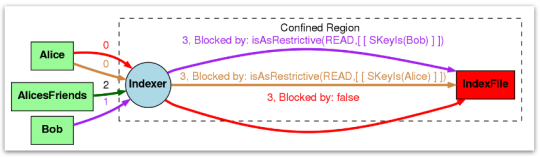


BLOCKING PREDICATES

false

isAsRestrictive(READ,[[SKeyIs(Alice)]])

isAsRestrictive(READ,[[SKeyIs(Bob)]])



Thanks to Bootstrap Themes

FIGURE B.2: Simplified search engine pipeline report 2

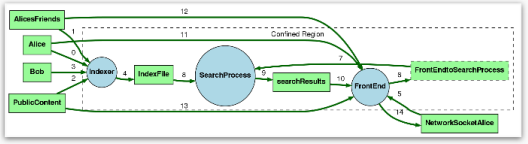
Appendix C

Simple search engine pipeline report

PolSim SUMMARY GRAPHKEY DECLASSIFICATIONS BLOCKERS

Policy Simulation Summary

THIS RUN WAS FLOWING!



IS Flowing!
END OF RUN#0_0

GRAPH KEY?

GRAPH KEY

Alice

Conduit [Alice,
Policy: [
Read: [[SKeyIs(Alice)]],
Update: [[SKeyIs(Alice)]],
Declassify: [[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])] V [Extrinsic & isAsRestrictive(READ,[[SKeyIs(Alice)]])]]]]]]
]

ConduitState: [SKeyIs : Alice]]

AlicesFriends

Conduit [AlicesFriends,
Policy: [
Read: [[FriendsOf(Alice)]],
Update: [[SKeyIs(Alice)]],
Declassify: [[[[[[isAsRestrictive(READ,[[FriendsOf(Alice)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])] V [Extrinsic & isAsRestrictive(READ,[[FriendsOf(Alice)]])]]]]]]
]

ConduitState: [SKeyIs : Alice]]

PublicContent

Conduit [PublicContent,
Policy: [
Read: [[true]],
Update: [[SKeyIs(Alice)]],
Declassify: null
]

FIGURE C.1: Simple search engine pipeline report part 1

ConduitState: [SKeyIs : Alice]]

Bob

Conduit [Bob,
Policy: [
Read: [[SKeyIs(Bob)]],
Update: [[SKeyIs(Bob)]],
Declassify: [[[[[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])] V [Extrinsic & isAsRestrictive(READ,[[SKeyIs(Bob)]])]]]]]]
]

ConduitState: [SKeyIs : Bob]]

Indexer

Process [Indexer, policies=
[[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])] V [Extrinsic & isAsRestrictive(READ,[[SKeyIs(Alice)]])]]]]]]&
[[[[[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])] V [Extrinsic & isAsRestrictive(READ,[[SKeyIs(Bob)]])]]]]]]]

IndexFile

Conduit [IndexFile,
Policy: [
Read: [[false]],
Update: [[true]],
Declassify: [[[[[[isAsRestrictive(READ,[[false]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]
]

ConduitState: []]

SearchProcess

Process [SearchProcess, policies=
[[[[[[isAsRestrictive(READ,[[false]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]]

searchResults

Conduit [searchResults,
Policy: [
Read: [[true]],
Update: [[ONLY_CND_IDS]],
Declassify: null
]

ConduitState: []]

FrontEnd

Process [FrontEnd, policies=
[[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])] V [Extrinsic & isAsRestrictive(READ,[[SKeyIs(Alice)]])]]]]]]]

FrontEndtoSearchProcess

Conduit [FrontEndtoSearchProcess,
Policy: [
Read: [[true]],
Update: [[true]],
Declassify: null
]

ConduitState: []]

NetworkSocketAlice

FIGURE C.2: Simple search engine pipeline report part 2

EConduit [NetworkSocketAlice,
Policy: [
Read: [[SKeyIs(Alice)]],
Update: [[true]],
Declassify: null
]

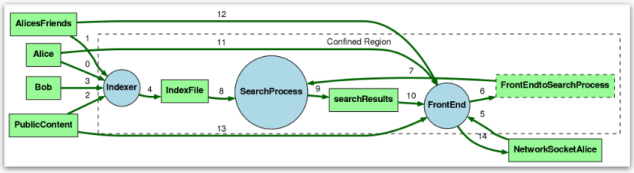
ConduitState: [ip : 192.168.1.1]]

SUCCESSFUL DECLASSIFICATIONS

searchResults

Conduit [searchResults,
Policy: [
Read: [[true]],
Update: [[ONLY_CND_IDS]],
Declassify: null
]

ConduitState: []]



NetworkSocketAlice

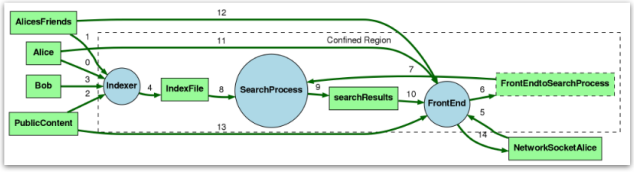
EConduit [NetworkSocketAlice,
Policy: [
Read: [[SKeyIs(Alice)]],
Update: [[true]],
Declassify: null
]

ConduitState: [ip : 192.168.1.1]]

BLOCKING PREDICATES

No Blockers!

This run was a success, thus no blocking predicates :)



Thanks to Bootstrap Themes

FIGURE C.3: Simple search engine pipeline report part 3

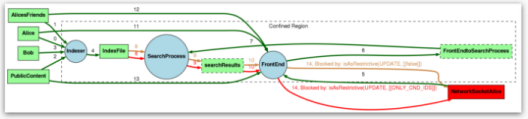
Appendix D

Simple search engine pipeline report - non compliant policies

PolSim SUMMARY GRAPHKEY DECLASSIFICATIONS BLOCKERS

Policy Simulation Summary

THIS RUN WAS BLOCKED!



Flow BLOCKING PREDICATES [isAsRestrictive(UPDATE,[[ONLY_CND_IDS]]), isAsRestrictive(READ,[[false]])]
++++++++++++++++++++++++++++

{Flow blocking predicate: isAsRestrictive(UPDATE,[[ONLY_CND_IDS]]) }
Flow blocking conduit: NetworkSocketAlice
Flow blocking predicate origin: IndexFile
Flow blocking Permission: [[[[[[isAsRestrictive(READ,[[false]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]
++++++++++++++++++++++++++++

{Flow blocking predicate: isAsRestrictive(READ,[[false]]) }
Flow blocking conduit: NetworkSocketAlice
Flow blocking predicate origin: IndexFile
Flow blocking Permission: [[[[[[isAsRestrictive(READ,[[false]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]
++++++++++++++++++++++++++++

Blocked!
END OF RUN#0_0

GRAPH KEY?

GRAPH KEY

Alice
Conduit [Alice,
Policy: [
Read: [[SKeyIs(Alice)]],
Update: [[SKeyIs(Alice)]],
Declassify: [[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]
]

ConduitState: [SKeyIs : Alice]]

FIGURE D.1: Simple search engine pipeline report part - non compliant policies 1

AlicesFriends

Conduit [AlicesFriends,
Policy: [
Read: [[FriendsOf(Alice)]],
Update: [[SKeyIs(Alice)]],
Declassify: [[[[[[isAsRestrictive(READ,[[FriendsOf(Alice)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]
]

ConduitState: [SKeyIs : Alice]]

PublicContent

Conduit [PublicContent,
Policy: [
Read: [[true]],
Update: [[SKeyIs(Alice)]],
Declassify: null
]

ConduitState: [SKeyIs : Alice]]

Bob

Conduit [Bob,
Policy: [
Read: [[SKeyIs(Bob)]],
Update: [[SKeyIs(Bob)]],
Declassify: [[[[[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]
]

ConduitState: [SKeyIs : Bob]]

Indexer

Process [Indexer, policies=
[[[[[[isAsRestrictive(READ,[[SKeyIs(Alice)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]&
[[[[[[isAsRestrictive(READ,[[SKeyIs(Bob)]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]]

IndexFile

Conduit [IndexFile,
Policy: [
Read: [[false]],
Update: [[true]],
Declassify: [[[[[[isAsRestrictive(READ,[[false]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]
]

ConduitState: []]

SearchProcess

Process [SearchProcess, policies=
[[[[[[isAsRestrictive(READ,[[false]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]]

searchResults

Conduit [searchResults,
Policy: [
Read: [[false]],
Update: [[true]],
Declassify: [[[[[[isAsRestrictive(READ,[[false]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]
]

ConduitState: []]

FIGURE D.2: Simple search engine pipeline report part - non compliant policies 2

FrontEnd

Process [FrontEnd, policies=
[[[[[[isAsRestrictive(READ,[[false]])]] UNTIL [[isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])]]]]]]

FrontEndtoSearchProcess

Conduit [FrontEndtoSearchProcess,
Policy: [
Read: [[true]],
Update: [[true]],
Declassify: null
]

ConduitState: []]

NetworkSocketAlice

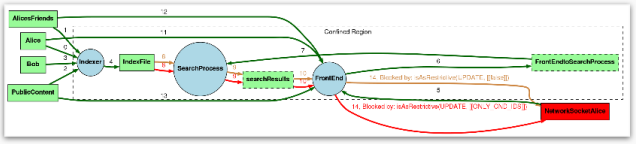
EConduit [NetworkSocketAlice,
Policy: [
Read: [[SKeyIs(Alice)]],
Update: [[true]],
Declassify: null
]

ConduitState: [ip : 192.168.1.1]]

SUCCESSFUL DECLASSIFICATIONS

Declassifications

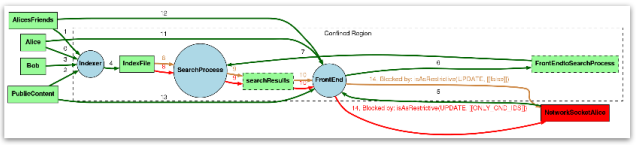
There were no successful declassifications



BLOCKING PREDICATES

isAsRestrictive(UPDATE,[[ONLY_CND_IDS]])

isAsRestrictive(READ,[[false]])



Thanks to Bootstrap Themes

FIGURE D.3: Simple search engine pipeline report part - non compliant policies 3

Appendix E

Coverage testing report

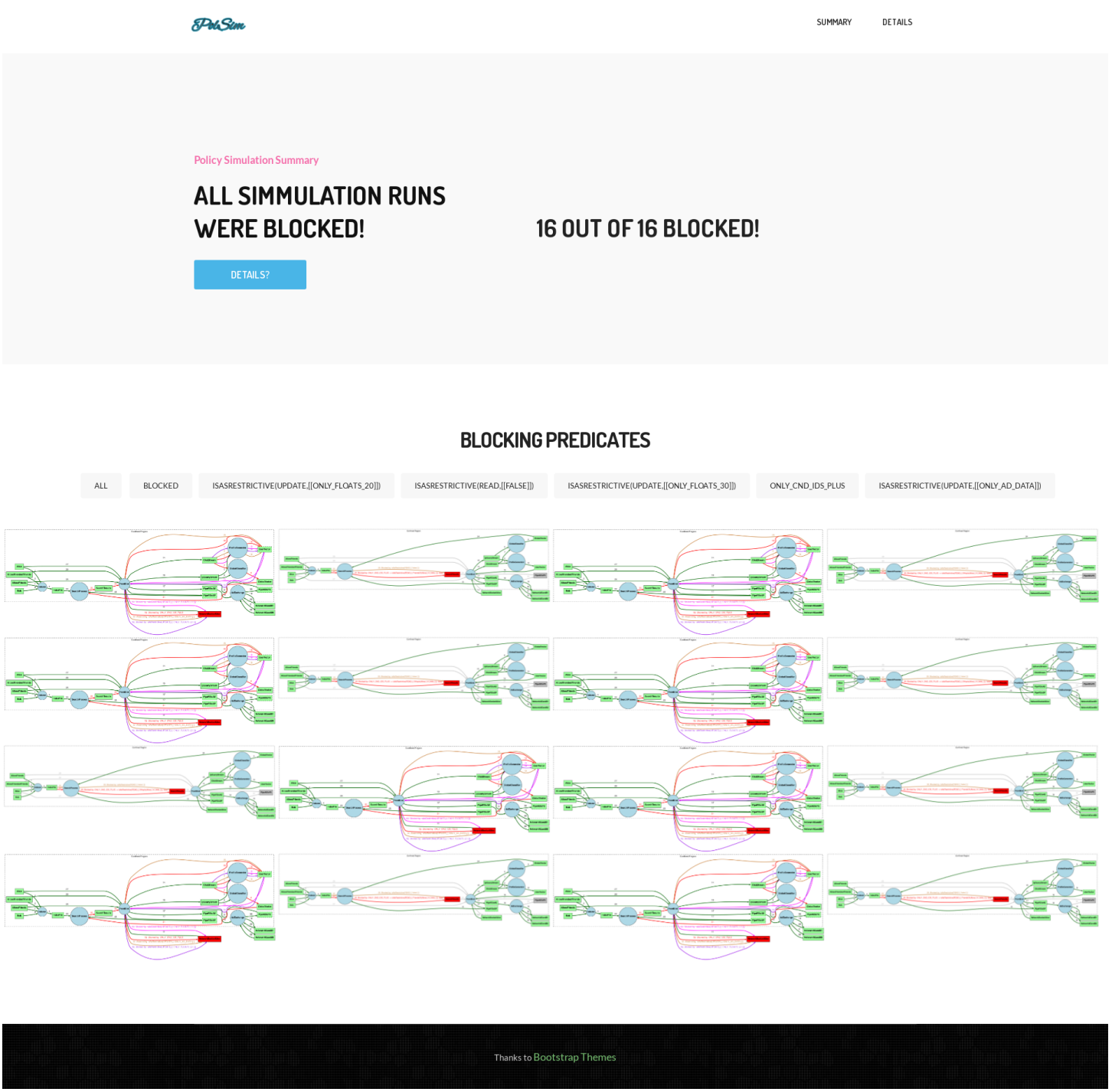


FIGURE E.1: Overview of all simulations

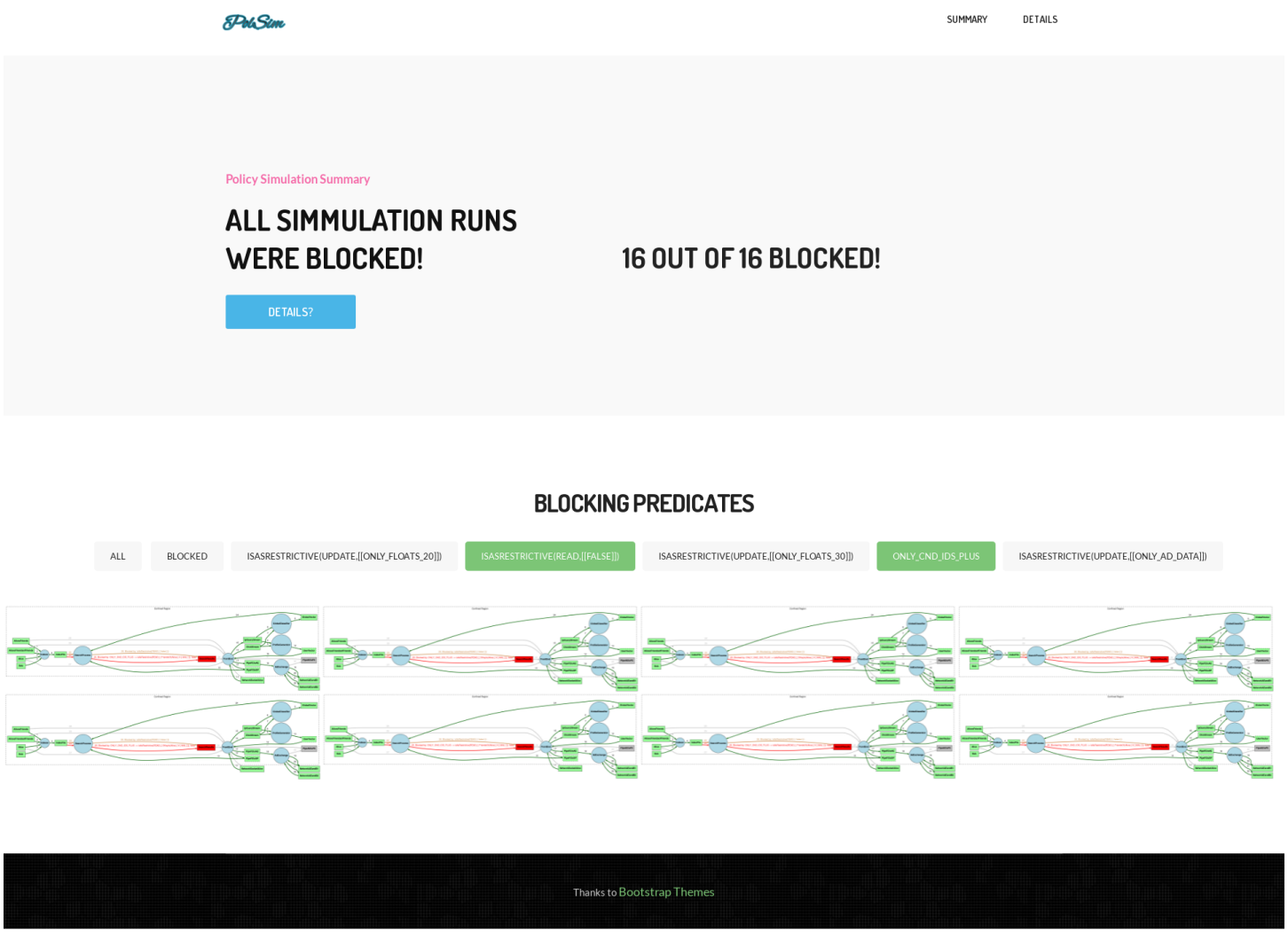


FIGURE E.2: Filtering with some predicates

Bibliography

[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] Adam Barth, John C. Mitchell, Anupam Datta, and Sharada Sundaram. Privacy and utility in business processes. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, 2007.

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

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

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

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

[16] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[17] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. Thoth: Comprehensive policy compliance in data retrieval systems. 2016.

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

[19] Deepak Garg, Limin Jia, and Anupam 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] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro 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] Katia Hayati and Martín 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] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control.

[23] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[24] Ninghui Li and John 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] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. Silverline: Preventing data leaks from compromised web applications. In *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

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

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

[28] TJ Parr and RW Quong. Antlr: A predicated. *Software—Practice and Experience*, 25(7):789–810, 1995.

[29] Frederick Ryckbosch, Stijn Polfliet, and Lieven 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, September 2006.

[31] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

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

[33] Dr. Kenneth Vollmar and Dr. Pete Sanderson. A mips assembly language simulator designed for education. *J. Comput. Sci. Coll.*, 21(1):95–101, October 2005.

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

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

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

[37] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Neon: System support for derived data management. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2010.