Enforcing Privacy Policies with Meta-Code

Håvard D. Johansen
UIT The Arctic Univ. of Norway
haavardj@cs.uit.no

Eleanor Birrell
Cornell University
eleanor@cs.cornell.edu

Robbert van Renesse
Cornell University
rvr@cs.cornell.edu

Fred B. Schneider
Cornell University
fbs@cs.cornell.edu

Magnus Stenhaug
UIT The Arctic Univ. of Norway
magnus@cs.uit.no

Dag Johansen
UIT The Arctic Univ. of Norway
dag@cs.uit.no

Abstract

This paper proposes a mechanism for expressing and enforcing security policies for shared data. Security policies are expressed as stateful meta-code operations; meta-code can express a broad class of policies, including access-based policies, use-based policies, obligations, and sticky policies with declassification. The meta-code is interposed in the filesystem access path to ensure policy compliance. The generality and feasibility of our approach is demonstrated using a sports analytics prototype system.

1. Introduction

Existing mechanisms lack effective means to express and enforce security and privacy policies on information after it has been shared. To provide that means, a mechanism must (1) support policies that can change depending on how data are manipulated, (2) apply policies to all copies of data and to any derived data, and (3) enforce policies wherever and whenever the original or derived data are used.

This paper explores how *meta-code operations* can express and enforce certain types of privacy policies even after data have been shared. Each data file is associated with some meta-code. The meta-code determines whether an access is permitted (as a function of, among other, the local meta-code state). Meta-code can also implement logging, provenance tracking, or other obligations. Meta-code logically resides in the filesystem access path; it is executed by the operating system to enforce the privacy policy.

APSys 2015, July 27–28, 2015, Tokyo, Japan.

http://dx.doi.org/10.1145/10.1145/2797022.2797040

By default, the same meta-code is automatically propagated to any derived data. Meta-code can override this default by explicitly specifying new meta-code that should be associated with derived data. For example, data recording an athlete's heart rate might be readable only by the data subject, but a coach might be allowed to view a fitness evaluation derived from that heart rate data.

Obviously enforcing privacy policies cannot be solved by technical means alone. Our work focuses on preventing accidental policy violations by well-intentioned users and on creating an audit trail that can establish accountability for policy violations.

2. Use Case: Elite Soccer Development

Elite soccer is a competitive domain in which technology has the potential to deliver a competitive edge [7]. Modern soccer clubs have systems that monitor players and collect data during training, during games, and even outside the sports arena. Sports scientists analyze data to personalize training, to prevent injuries, and to provide a foundation for evidence-based decisions about team performance improvements.

Collected data are stored in credential-protected data stores. Data can be collected either explicitly or implicitly. Explicit data are provided directly by individual athletes (e.g., entered into an app on a mobile device) and might include information on perceived fatigue, sleep quality, sleep duration, mood, or muscle soreness. Implicit data are collected automatically from sensors (e.g., Fitbits, wearable sensor arrays, or video cameras) and might include positional data, body direction, or heart rate while exercising. In prior work, we developed and deployed prototype systems for elite Norwegian soccer clubs [17]. This work includes Muithu, a notational analytics system that allows coaches to make notations and provide video feedback in real-time, and Bagadus, a system that integrates radio sensor positioning, soccer analytics annotations, and a video camera array with real-time processing

There are security and privacy constraints related to athlete and team data. In order to comply with athletes' privacy preferences and with relevant legal frameworks, data must be handled with care by the principals that access and manipulate it. These principals include athletes, academy players, sports scientists, physical trainers, medical staff, psychologists, nutritional scientists, main coaches, administrative staff, journalists, and supporters. Mechanisms must therefore enforce security and privacy policies in a heterogeneous and diverse setting.

To simplify this example use case, we associate users with roles; each role has privileges defined by the data owner.

Athlete. Athletes are owners of any data collected about them (whether explicit and implicit).

Medic. Medics are team employees who have access to certain raw and derived data about athletes. Medics provide health care and recommendations to athletes based on that data.

Coach. Coaches have access to recommendations from medics and to certain information inferred from raw data. Coaches make decisions about training schedules and game rosters on the basis of this derived data.

Public. The public includes team fans and media who appreciate statistics and trivia about their favorite team.

By default, anyone other than the data owner should be denied access to collected data; the owner must explicitly grant permissions to a role before users in that role can view the data.

The following is a simple example privacy policy governing how *data-store credentials* and the information derived from them may be used. Credentials retain strict access permissions.

POLICY RULE 1. *Only the data owner is authorized to view credentials for a private store containing raw sensor data (e.g., an OAuth2 token for a Fitbit account).*

Although a strict policy is appropriate for credentials, data retrieved with these credentials might have more relaxed permissions. For example, a medic might need to access recent sensor data to diagnose an illness or injury.

POLICY RULE 2. *Medics are authorized to view raw fitness data collected within the last 24 hours.*

Coaches also need information about athlete fitness in order to decide about individual training regimes and about the team roster for upcoming games. Coaches do not need access to detailed raw data, although they do need to make decisions on the basis of the most up-to-date data available.

POLICY RULE 3. *Coaches are authorized to view smoothed athlete data. For example, a coach may see a readiness-to-train score calculated from medical recommendations, Fitbit sleep data, and perceived training load.*

Principals outside the team staff (e.g., fans and sports reporters) are interested in information and trivia about athletes—for example, how far each athlete ran during a particular game. However, raw data (e.g., precise GPS locations) and information derived from certain sensitive values (e.g., heart rate or sleep patterns) should not be available to the public.

POLICY RULE 4. *De-sensitized, smoothed statistics derived from athlete data may be released to the public.*

3. The LoNet Architecture

LoNet is a runtime system that augments data with mandatory and discretionary security policies. LoNet runs inside a machine virtualization container (e.g., VirtualBox[1] or Docker[2]) and implements a reference monitor that executes small code snippets called *meta-code* on data accesses. LoNet is compatible with programs written in any programming language.

Our current LoNet prototype adopts the file system as its primary interface.[3] LoNet stores all data in files, and files are organized in hierarchical directories. Each file can be associated with meta-code that implements a data policy. For example, meta-code might check that a principal is a medic before permitting access to raw heart-rate data, or it might create a log entry each time a coach accesses a smoothed data file. Meta-code can also contain state, so LoNet supports policies like "medics can access fitness data only within 24 hours of data capture." Policies are defined at the granularity of full files in order to limit the burden imposed when expressing a policy.

Meta-code for a file can be defined by the file's owner at the time the file is created. Meta-code can also be added subsequently, by a data processor. In addition, any newly created file inherits meta-code from files on which the content of the new file depends. Such inherited meta-code need not be the same as the meta-code associated with the original files; instead inherited meta-code might be a new code snippet specified by the original meta-code of the files read. Inherited meta-code is specified as a function of the type of program used to create the derived data file. For example, meta-code associated with a raw data file might only allow the data owner to read that file; meta-code for a file containing readiness-to-train scores (derived from the raw data) could additionally allow reads by coaches.[4]

[1] `https://www.virtualbox.org/`

[2] `http://www.docker.com/`

[3] Although we do not directly support databases or structured files, fine-grained policies could be implemented by employing an appropriate file-based data model for databases or by otherwise subdividing data into many small files.

[4] By default, derived files inherit the meta-code associated with the original files; in this case, meta-code inheritance reduces to standard taint tracking.

Using LoNet, a user can share a file (and the associated meta-code) with other users. LoNet will permit the receiver to view the file only if the receiver acts in an authorized role. A receiver can also execute programs that read a received file. A program is authorized to read a file if either (1) it is invoked by a user who is authorized to read that file or (2) the program is of a type that is authorized to read that file. If meta-code specifies inherited meta-code to be associated with the output of some program type, then programs of that type are authorized to read the file. For example, an athlete can share Fitbit credentials (associated with meta-code that implements the policy described in Section 2) with a coach. LoNet will not permit the coach to view the credentials, but the coach may run a program of type project that uses the credentials to download data from the athlete's Fitbit data store. The inherited meta-code means LoNet will not permit the coach to view the downloaded data file, but if the coach runs a smoothing function over the downloaded data, the resulting (smoothed) data file may be viewed.

We have implemented a prototype of LoNet using the FUSE user-level file system library, available in Linux 2.6 and Linux 3 kernels, in combination with VirtualBox containers. The LoNet daemon process runs as a privileged system user inside the container; it has full access to a hidden source file-system directory. The source directory is exported to a known mount point accessible to system processes. The authorizations of those system processes are controlled by LoNet. The LoNet daemon intercepts all file system calls to the visible mount point and invokes meta-code execution and propagation routines to enforce and propagate the security policies attached to files. Users acting in roles interact with protected objects within the LoNet sandbox through traditional file system abstraction and tools.

3.1 Meta-code

Each data file is associated with a *policy file*—an auxiliary file that is normally hidden from users. A policy file contains (1) a list of roles permitted to access the associated data file, (2) a mapping from hooks to code snippets that implement the meta-code that should be run when various events occur, like file access, and (3) a mapping from *transition types*—classes of programs—to policy files specifying which policy file to associate with a derived file. If a derived file is created by a program that does not match any of the specified transition types, then the derived file inherits the policy file associated with the input file. Figure 1 shows example policy files implementing the policy specified in Section 2. Figure 2 presents the meta-code referenced in Figure 1b.

The descendants of the policy file associated with a raw data file induce a *policy automaton* [2, 13] that specifies the meta-code that should be associated with any derived data file. Figure 3 shows the policy automaton for the policy from Section 2. Each box corresponds to a policy file from Figure 1 and lists the roles authorized to view a file associated with that policy; the box labeled "start" corresponds to the file `/pol/priv`. Arrows indicate which policy files is associated with derived files generated by programs of that transition type.

Data owners and data processors both may associate meta-code with files using the *extended file attributes* feature available in modern Linux kernels and filesystems. To associate meta-code with a file, a principal uses the `setfattr` command. For example, if `credentials` is the file containing the Fitbit OAuth2 token and `/pol/priv` is the policy file given in Figure 1a, then an athlete can express the policy from Section 2 with the command

```
#> setfattr -n policy -v /pol/priv credentials
```

The credential file will be associated with the policy file `/pol/priv` which specifies that only a LoNet container run by the credential owner may view that file. Files derived from the credential file inherit meta-code determined by the type of program that created them. For example, if a medic (who cannot view the credential file) runs a program of type project that uses the credentials file to download raw data from the Fitbit data store, the downloaded data file will be associated with the policy file `/pol/raw` defined in Figure 1b. The policy on the downloaded data enables the medic to initially read the output data. However, this policy specifies that the `/code/24hr` meta-code should run on each file access. The meta-code, shown in Figure 2, compares the file creation time with the current time and returns the `EACCESS` error if the difference is more than 24 hours.

3.2 Users

Meta-code can depend on which user triggered the event or on which role(s) are associated with that user. FUSE therefore needs to inform the meta-code about which principal is accessing the file system, and in which role. Unfortunately, the FUSE interface only provides Linux process-id, user-id, and group-id of the calling process. To circumvent this problem, LoNet requires that user certificates are passed as part of the file name itself, as the first component of a Linux path name.

3.3 Program Transition Types

Policy file transitions are specified in terms of classes of programs called *transition types*, or ttypes. A trusted authority defines ttypes and specifies a mapping between programs and ttypes.

In LoNet, a ttype is defined by a pair comprising the hash of a program binary and an expression that restricts arguments of that program. For instance, a ttype might require that the first positional argument in a file must match some specified hash value. Restrictions on arguments enable LoNet to map programs expressed in scripting languages, like Python or R, based on both the binary and the script file being loaded. These ttype definitions are specified in a configuration file, as illustrated in Figure 4.

```
[permissions]
roles = {}

[transitions]
project=/pol/raw
```

(a) /pol/priv

```
[permissions]
roles = {medic}

[metacode]
onAccess=/code/24hr

[transitions]
smoothing=/pol/smoothed
desensitize=/pol/desens
```

(b) /pol/raw

```
[permissions]
roles = {coach}

[transitions]
desensitize=/pol/pub
```

(c) /pol/smoothed

```
[permissions]
roles = {medic}

[transitions]
smoothing=/pol/pub
```

(d) /pol/desens

```
[permissions]
roles = *
```

(e) /pol/pub

Figure 1: Example policy files that express the policy defined in Section 2.

```
import errno, time, os
from fuse import FuseOSError

# setup by LoNET: path

if (time() - lstat(path).st_ctime) > 86400:
raise FuseOSError(EACCES)
```

Figure 2: The meta-code /code/24hr that allows file access only within 24 hours.

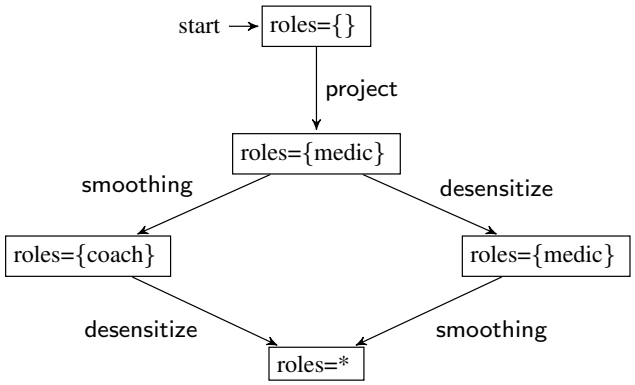

Figure 3: A graphic depiction of the policy automaton described by the policy files in Figure 1.

Our ttype specification currently uses SHA-256 for hashing executables. Attested program binaries are identified by their SHA-256 hash, and specified by the `exe` value in the program's configuration. Alternatively, a program can be specified by a `exe_path` value to a trusted binary location. In this case, LoNet computes and sets the program's `exe` value on startup.

To find the appropriate ttype of a file access, LoNet maps the process-id to the appropriate ttype. This is done by (1) reading and hashing the executable for process-id as provided through `/proc/{pid}/exe`; (2) reading and parsing the program arguments as provided through `/proc/{pid}/cmdline`; and (3) matching the hash of the executable and its arguments to a ttype as specified in the ttype configuration file.

```
[cat]
exe_path = /bin/cat
ttype = publish

[Analyze]
exe=e671...
options_match = -B -d -E -h -i -O -OO ...
options_havearg = -m -Q -W -c
arg0_type = hash
arg0 = 0x[...]17011
ttype=desensitize

[Download]
exe=e671...
options_match = -B -d -E -h -i -O -OO ...
options_havearg = -m -Q -W -c
arg0_type = path
arg0 = /code/download
ttype=project
```

Figure 4: Example ttype specification file.

LoNet maintains a cache of recently seen pids and their ttype, which avoids some of the cost related to the mapping of process-ids to ttype. A program can still switch between ttypes by spawning new processes.

3.4 Sessions

In order to associate the appropriate inherited meta-code with derived files, FUSE needs to know the set of files on which a derived file depends. To support such policy inheritance while supporting automation tools and interactive programs—like GNU Make or bash-shells—that might invoke multiple other sub-programs, LoNet implements *sessions*. Each program is run in the context of a session, and each session is associated with a principal. Sessions are used to implement meta-code propagation; each time a program accesses a data file, the session is tainted with the policy file associated with that data file. The policy for an output file is the conjunction of the policy files in the session taint. By propagating taint exclusively within a session, LoNet limits over-tainting.

A program initiates a new session by reading from dedicated file `/newsession`—which returns an unique 32 B session identifier—and setting the SESSION_ID environment variable to the returned session identifier prior to the program's first file operation. Programs that do not explicitly specify a session identifier are mapped to a default NULL session; programs run in the context of the default NULL session might be associated with unnecessary meta-code.

4. Evaluation

To quantify the overhead of the policy-enforcing mechanism in LoNet, we evaluated a set of micro-benchmarks. We ran all experiments on a Lenovo Thinkpad T430s, sporting an Intel Core i7-3520M CPU with four cores, each running at 2.90 GHz; 12 GB of main memory; and a 500 GB Samsung 840 EVO SSD drive. This hardware is representative of what our expected end-user would have available to them. The machine runs 64-bit Ubuntu 13.04 and we use Anaconda Python 3.4 to run LoNet.

First, we measure how IO performance is affected by our meta-code propagation and policy transition mechanisms by measuring the time it takes a Python script to copy data between two files within the LoNet container. The script reads and writes using 32 kB blocks; the initial files contain random bytes. Figure 5 shows the observed overhead of LoNet on files with and without meta-code policies for file sizes ranging from 1 kB to 64 MB. For comparison, we also run the experiment directly on the underlying Ext4 filesystem, and on a minimal Python-based FUSE filesystem (Python + FUSE), which only passes invoked operations to the underlying filesystem. Each experiment is repeated 10 times; the figure reports the mean. As shown in Figure 5, there is significant overhead associated with LoNet. This overhead is mostly due to overhead imposed by having a userspace filesystem and employing an interpreted language. We hypothesize that a kernel implementation of LoNet would eliminate much of that overhead. Also, our current implementation of LoNet executes the full policy transition mechanism and policy propagation on every read. Significant performance improvement would result from optimizing cases where session taint has not changed between consecutive file operations.

Figure 6 reports how block size affects our copy experiment. We set the file size to 64 MB and vary the block size between 64 B and 64 kB. As expected, the increase in IO operations due to a small block size adversely impacts LoNet's performance.

Finally, we evaluate how LoNet performs when aggregating a large number of files. For this, we implement a Python script that consecutively reads and computes the SHA-256 hash for a given set of files. Figure 7 shows the time it take to aggregate between 1 and 1000 files. Each test file contains 10 MB of random data. The block size is set to 16 kB.

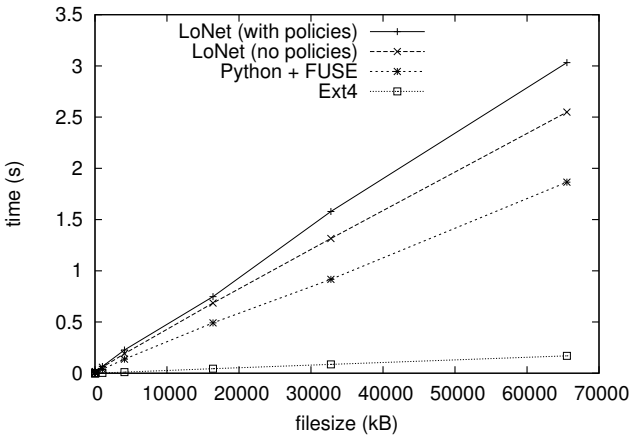

Figure 5: Copy performance with filesize.

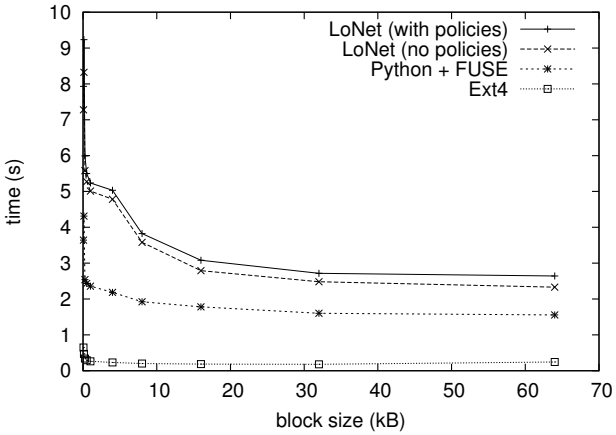

Figure 6: Copy performance with blocksize.

As shown in Figure 7, computational intensive workloads are less affected by the IO overhead of LoNet.

5. Related Work

Information Flow Control. First introduced by Denning [5], Information Flow Control (IFC) concerns how information may and may not flow through a system. These restrictions are often specified through labeling [15], whereby data is (logically or physically) marked with a security level that is propagated as the data is being manipulated. IFC can be enforced using static language-based methods, as demonstrated by the security-typed Java derived programming language JiF [18]. Here, programmers attach labels on variables at the source-code level to express restriction on information flow within the program. TaintDroid [9] takes a different approach to IFC by extending Google's Android kernel to track third-party binary applications dynamically at run-time by labeling data from possible sensitive sources as tainted during execution. Later work [1, 3] has adapted the concept of information flow to express policies about derived values.

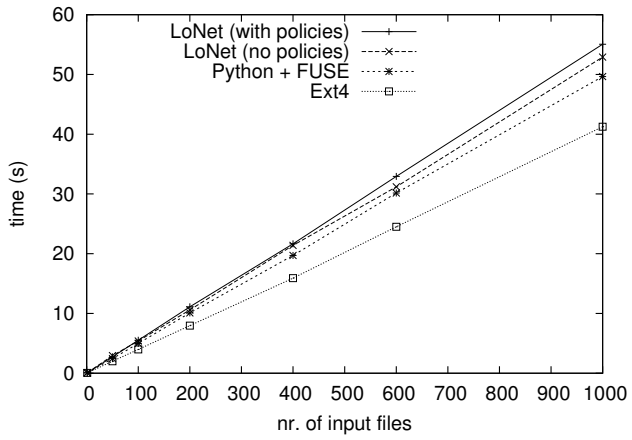

Figure 7: File aggregation performance.

Privacy Policy Enforcement. There are many existing approaches to enforcing privacy policies. Garg et al. [10] describe a method for detecting policy violations (expressed in a first-order logic) given a set of partial audit logs. Secure Data Capsules [14] are data objects associated with a policy tag that defines the provenance and a usage policy. These policy tags are cryptographically bound to the associated data, and derived data are tagged with a derived policy tag. Untrusted applications can operate on tagged objects only while inside a secure execution environment that enforces and propagates policy tags. Sen et al. [19] statically enforce privacy policies in large codebases by analyzing data flow graphs constructed automatically from minimally-annotated code. Guardat [20] enforces policies about reading and writing files at the storage layer by comparing credentials against its own meta-data for policy-protected files.

Sandboxing. Although existing sandboxing techniques are effective in preventing leakage of sensitive data, execution isolation alone can be too restrictive because permitted data flow into and out of the sandbox is difficult to express. Vanhoef et al. [22] argue that data processed in a web-browser's JavaScript sandbox can safely be declassified using a combination of idempotent projection functions in combination with stateful release functions, both attached to events in the runtime. The proposed machinery has been implemented in FlowFox as a Firefox extension that executes declassification policies on invocation of the available JavaScript event handlers. The declassification policies are enforced using Secure Multi-Execution (SME) techniques [6], which can become computationally expensive when the number of taint labels increases.

Meta-code. Meta-code [11] extends a two decade long foray into mobile code as a structuring mechanism in distributed systems. Our early TACOMA mobile agent system was used for management purposes by inter-positioning mobile agents in the access path of data [12]. Our recent Codecaps protocol adds executable code fragments in cryptographically protected capabilities to enable flexible discretionary access control in cloud-like computing infrastructures [21]. And, we have already used meta-code for building a distributed storage system [16] that spans multiple, heterogeneous clouds and involve data with confidentiality constraints. Our meta-code programming model is conceptually similar to the active document properties in the Placeless system [8] and the monitoring-oriented programming proposed by Chen and Roşu [4]. However, LoNet takes advantage of more recent development in IFC methods and container technologies.

6. Conclusion and Future Work

LoNet is an architecture that enables expressive data privacy policies attached to files in the form of programmable meta-code; enforcement is achieved by intercepting file system operations. The meta-code data policies can implement finite-state automata that transition between states on file access. LoNet enables data sharing and analysis while imposing access control restrictions for both raw and derived data. We demonstrate the feasibility of the approach by implementing a policy motivated by data analytics used in elite soccer development.

In order to increase the level of security, we are porting our FUSE-based framework to a secure container system; we are currently experimenting with Intel Software Guard Extensions (SGX) for this purpose. In addition, we would like to enable data sharing between mutually trusting containers by leveraging remote attestation. We are also working on a kernel implementation for improved performance.

Acknowledgments

This work was supported in part by the Norwegian Research Council project number 231687/F20, by AFOSR grants F9550-06-0019 and FA9550-11-1-0137, by National Science Foundation grants 0430161, 0964409, and CCF-0424422 (TRUST), ONR grants N00014-01-1-0968 and N00014-09-1-0652, and grants from Microsoft. We would like to thank our shepherd Saikat Guha and the anonymous reviewers for their useful insights and comments.

References

[1] S. Bandhakavi, C. C. Zhang, and M. Winslett. Super-sticky and declassifiable release policies for flexible information dissemination control. In *Proc. of the 5th ACM Workshop on Privacy in Electronic Society*, WPES '06, pages 51–58. ACM, 2006. doi: 10.1145/1179601.1179609.

[2] E. Birrell and F. Schneider. Fine-grained user privacy from avenance tags. Unpublished manuscript, 2015.

[3] L. Bussard, G. Neven, and F.-S. Preiss. Downstream usage control. In *Proc. of the IEEE Internation Symposium on Policies for Distributed Systems and Networks*, POLICY '10,

pages 22–29. IEEE, July 2010. doi: 10.1109/POLICY.2010.17.

[4] F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. *ACM SIGPLAN Notices - Proc. of the 2007 OOPSLA conference*, 42(10):569–588, Oct. 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297069.

[5] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976. doi: 10.1145/360051.360056.

[6] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124. IEEE, May 2010. doi: 10.1109/SP.2010.15.

[7] P. Dizikes. Sports analytics: a real game-changer. MIT News export-mar 4, Massachusetts Institute of Technology, Mar. 2013.

[8] P. Dourish, W. K. Edwards, J. Howell, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. Terry, and J. Thornton. A programming model for active documents. In *Proc. of the 13th Annual ACM Symposium on User Interface Software and Technology*, UIST '00, pages 41–50. ACM, 2000. ISBN 1-58113-212-3. doi: 10.1145/354401.354410.

[9] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–6. USENIX Association, 2010.

[10] D. Garg, L. Jia, and A. Datta. Policy auditing over incomplete logs: Theory, implementation and applications. In *Proc. of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 151–162. ACM, 2011. doi: 10.1145/2046707.2046726.

[11] D. Johansen and J. Hurley. Overlay cloud networking through meta-code. In *Proc. of the 35th IEEE Annual Computer Software and Applications Conference Workshops*, COMPSACW '11, pages 273–278. IEEE, July 2011. doi: 10.1109/COMPSACW.2011.54.

[12] D. Johansen, K. Marzullo, and K. J. Lauvset. An approach towards an agent computing environment. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems Workshop*, ICDCS '99. IEEE, May 1999. doi: 10.1109/ECMDD.1999.776418.

[13] E. Kozyri and F. Schneider. Reactive information flow specifications. Unpublished manuscript, 2014.

[14] P. Maniatis, D. Akhawe, K. Fall, E. Shi, S. McCamant, and D. Song. Do you know where your data are? Secure data capsules for deployable data protection. In *Proc. of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS '11, pages 22–27. USENIX Association, 2011.

[15] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, Oct. 2000. ISSN 1049-331X. doi: 10.1145/363516.363526.

[16] A. Nordal, Å. Kvalnes, J. Hurley, and D. Johansen. Balava: Federating private and public clouds. In *Proc. of the 9th IEEE World Congress on Services*, SERVICES '11, pages 569–577. IEEE, July 2011. doi: 10.1109/SERVICES.2011.21.

[17] S. A. Pettersen, D. Johansen, H. Johansen, V. Berg-Johansen, V. R. Gaddam, A. Mortensen, R. Langseth, C. Griwodz, H. K. Stensland, and P. Halvorsen. Soccer video and player position dataset. In *Proc. of the 5th ACM Multimedia Systems Conference*, MMSys '14, pages 18–23. ACM, Apr. 2014. doi: 10.1145/2557642.2563677.

[18] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003. ISSN 0733-8716. doi: 10.1109/JSAC.2002.806121.

[19] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing. Bootstrapping privacy compliance in big data systems. In *Proc. of the 35th IEEE Symposium on Security and Privacy*, SP '14, pages 327–342. IEEE, May 2014. doi: 10.1109/SP.2014.28.

[20] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post. Guardat: Enforcing data policies at the storage layer. In *Proc. of the 10th European Conference on Computer Systems*, EuroSys '15, pages 13:1–13:16. ACM, 2015. doi: 10.1145/2741948.2741958.

[21] R. van Renesse, H. D. Johansen, N. Naigaonkar, and D. Johansen. Secure abstraction with code capabilities. In *Proc. of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '13, pages 542–546. IEEE, Feb. 2013. doi: 10.1109/PDP.2013.87.

[22] M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *Proc. of the 27th IEEE Computer Security Foundations Symposium*, pages 293–307. IEEE, July 2014. doi: 10.1109/CSF.2014.28.