Privacy Linting to Statically Find Privacy Violations

Justus Adam
me@justus.science
Brown University
Providence, RI, USA

Livia Zhu
livia_zhu@brown.edu
Brown University
Providence, RI, USA

Sreshtaa Rajesh
sreshtaa_rajesh@brown.edu
Brown University
Providence, RI, USA

Malte Schwarzkopf
malte@cs.brown.edu
Brown University
Providence, RI, USA

Will Crichton
will_crichton@brown.edu
Brown University
Providence, RI, USA

Shriram Krishnamurthi
shriram@brown.edu
Brown University
Providence, RI, USA

Abstract

Organizations struggle to ensure that their developers' code adheres to privacy requirements. We propose a *privacy linter*, a new kind of program analysis tool that helps developers check that software conforms to privacy policies. A privacy linter must relate abstract, high-level privacy properties to concrete, frequently-changing source code without imposing rigid requirements on code structure. It must also provide useful errors to developers, and would ideally suggest repairs to source code. We describe a privacy linter design based on dependency analysis and model checking that captures common privacy properties and leverages the Rust type system. We apply our prototype, Paralegal, to a practical homework submission system.

1 Introduction

Even within a single company, the number of developers modifying a shared codebase on a daily basis makes it difficult to correctly implement and adhere to the organization's self-imposed privacy policies [10, 33]. Today, organizations must rely on manual audits by privacy experts or external consultants to check if their code respects important privacy properties. Such properties might include users' right to access and delete their data under the GDPR, time-limited data retention, or the confidentiality of sensitive data. Naturally, manual audits are laborious, error-prone, and unlikely to happen frequently [23, 26].

We propose a *privacy linter*, a new tool that warns developers of potential privacy problems they introduce during development. We imagine that engineers would regularly run the privacy linter on their codebase during the software development process. This would flag some privacy issues outside the audits so that developers can remove them *before* deployment.

Designing the ideal privacy linter will require solving several key research problems:

(1) Traditional linters tend to check syntax-level code patterns, but this is insufficient for detecting privacy issues, which requires semantic understanding of code.
(2) Privacy policies vary between organizations, countries, and across time, so a privacy linter must support a flexible and expressive range of policies.
(3) A strong privacy linter should ideally go beyond diagnostics to also propose potential fixes.
(4) Using a privacy linter should necessitate minimal changes to source code.

In this paper, we demonstrate the feasibility and potential utility of privacy linting by exploring one point with appealing qualities in the broader design space. We structure our discussion around three components which we believe are useful to a privacy linter that meets the above criteria. First, we use static analysis to obtain data and control-flow dependencies. Second, lightweight annotations on the source code provide semantic meaning to the code elements. The annotation burden is small: only functions and values *explicitly relevant to the policies* need annotations. Finally, we use a model-checker to verify a flexible set of privacy policies and a solver to generate repair suggestions.

We realize these ideas in Paralegal, a prototype privacy linter that targets Rust programs. Paralegal is built atop Alloy [13], a rich specification language with a corresponding SAT solver. We present preliminary findings of applying our prototype to a homework submission application deployed at a U.S. university and open avenues for further research to make privacy linting a reality.

2 A Motivating Example

The GDPR stipulates that users must be able to request that a system "forget" their personal data. Failing to provide a deletion option that deletes *all* of a user's data has led to severe consequences for companies, including the imposition of fines of up to €8M [1, 24, 25].

```
1  Found violation of rule:
2      `right_to_be_forgotten'.
3  Caused by a call in line 8:
4      8: database.insert("grades", student_record);
5  Relevant context that influenced this call:
6      7: let student_record = vec![user.id, grade];
7  This rule can be satisfied by removing the call:
8      8: database.insert("grades", student_record);
9  Or by inserting the call:
10     delete_grade(grade);
11 In the function:
12     15: forget_user()
```

Listing 1: *An ideal error message from a privacy linter.*

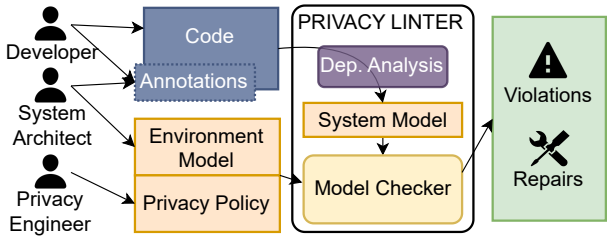

Figure 1: *A privacy linter checks annotated application code against a privacy policy with the aid of an environment model and a system model that captures program dependencies. It reports possible privacy violations and suggests repair options to developers.*

Consider Websubmit, a homework submission system [3] implemented in Rust using the Rocket web framework. Because Websubmit must follow the GDPR, a developer implements a deletion endpoint, `forget_user`. This endpoint deletes a user's personal data, such as their submitted answers. Later, the developer adds functionality to allow instructors to store student grades in the database. Under time pressure, they forget to update the `forget_user` endpoint to also delete grades. This is a prime example of an issue that we would like a privacy linter to catch.

To begin, a privacy team formulates the GDPR's right to be forgotten with respect to this application in the form of a property (we discuss this more in §3): *There exists one HTTP endpoint, accessible to each site user, which invokes a deletion method on every item of every type of personal data stored by the application about that user.*

The developer then runs the privacy linter to check whether they have introduced any privacy problems that would break this rule. The linter analyzes the code and observes that there now exists a stored personal type (`Grade`) that never gets deleted. It then produces the error message in Listing 1 that identifies the problem and suggests a fix: calling the `delete_grade` function in `forget_user`.

Of course, there are other properties that a privacy team would want to enforce, including access control, secure storage, purpose limitation, data deletion, temporal restrictions and obligations on data, which a privacy linter should also support.

3 Designing a Privacy Linter

An effective privacy linter needs to extract relevant information from the program, while asking for as little input as possible from the developer. We expect many kinds of information to be explored in future research on privacy linting, but we focus our discussion here on *program dependencies*.

Dependencies are highly relevant to privacy linting because many privacy properties can be approximated in terms of whether certain data is required to or prohibited from reaching some program location. For example one could approximate GDPR-style data deletion as *requiring* all data associated with a user to reach a deletion function in a deletion endpoint; an encryption-at-rest mandate as *requiring* all data to reach an encryption function *before* reaching a storage function; and access control as *prohibiting* sensitive user data from reaching functions that share it with unauthorized parties. While program dependencies are insufficient to *prove* compliance, our hypothesis is that linting for these weaker properties will uncover sufficiently many non-compliant programs during development to prove useful.

Extracting precise dependency information is challenging. For most languages, dependency analysis is either prohibitively overapproximating or resource intensive. In contrast, a restrictive yet practical memory model like Rust's ownership facilitates both precision and efficiency, making a privacy linter feasible in practice.

Overview. To begin, a privacy engineer would create a machine-readable formalization of the privacy properties they intend to check. A system architect would then create an environment model that provides additional system-specific context such as the access control policy. Finally, the developer annotates select source code elements with relevant semantic information. Then, the linter's work begins.

The privacy linter first extracts dependency graphs from the source code which comprise the *system model* (Figure 1). To analyze the graphs, modern model-checkers are a good choice, because they support a diverse set of properties. Furthermore, modern SAT or SMT solvers can reason about possible changes to the graph, which the linter can use to localize errors and make repair suggestions.

System Model. The system model is an abstraction over the program code that captures data and control flow, but eschews the lower-level details of the code. Specifically, we believe that a graph of data- and control-flow dependencies

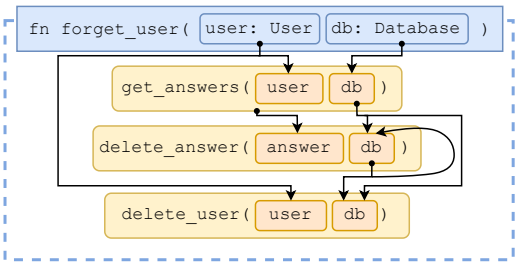

Figure 2: *Dataflow graph for the* `forget_user` *function, showing data flowing between request inputs, values from function calls and arguments.*

between function call sites is a helpful abstraction for a privacy linter. Function calls are a natural granularity choice as functions often represent semantically meaningful units. Consider the following function, which deletes a user in Websubmit:

```
fn forget_user(user: User, db: Database) {
    for answer in get_answers(&user, &mut db) {
        delete_answer(answer, &mut db);
    }
    delete_user(user, &mut db);
}
```

Figure 2 shows the *data dependency graph* calculated for this function. Historically, such an approximation has had to be conservative with respect to pointers; however, Rust's restrictive borrow semantics allow more precise reasoning.

Consider the two parameters, `user` and `db`. Because `user` is passed *immutably* (the default in Rust) to `get_answers`, a separate edge points from `user` to `delete_user`, indicating that both will see the same value. In contrast, `db` is passed *mutably* to `get_answers`. Therefore, subsequent uses of `db`, like `delete_answer`, receive a potentially modified value of `db` from `get_answers`.

Finally, an expressive system model should also contain a *control-flow dependency graph*, which tracks what values influence *how often* a function call is executed. This helps the linter reason about function calls under conditionals and implicit flows.

Labels. We expect privacy properties to be formally specified by privacy engineers, independent of day-to-day development. However, these properties will need to reason about program semantics. While data and control dependency analysis provides general-purpose semantics, we imagine system architects could define a set of lightweight *labels* for developers to annotate relevant parts of their code to provide *domain-specific semantics*. This would allow privacy properties to talk about more abstract notions of, say, "a deletion function" or "personal data," instead of brittle program constructions such as concrete call sites, types, and functions that are prone to frequent change. The example in §2 might assign the `personal` label to the `Grade` type, and apply the `deletes` label to the `Grade` argument of the `delete_grade` function. As opposed to classic information flow control (IFC) labels, which are constrained, e.g., to predetermined security levels, our annotations provide a *flexible* method to express a system-specific ontology for privacy properties.

Locating Privacy Issues. The system model is handed to the model-checker, which checks it against a set of privacy policies. For any policies that the model fails to satisfy, the linter invokes a solver to find the minimal set of edges in the system model's dependency graphs that must be removed to satisfy the policy. These edges effectively locate the error in the source code, and are displayed to the user along with additional context in an error message. This error message also contains a set of repair suggestions, discussed next.

Repair Suggestions. While just obtaining an indication of a possible privacy problem is already valuable, developers benefit even more from *actionable* feedback. We identify the solver as a key tool that can explore possible modifications to the system model that fix the property, without needing to understand the semantics of the system. Using a set of guiding constraints, the privacy linter queries the solver for different possible deltas to the system model that could fix the property violation. These suggestions include but are not limited to: removing erroneous data flows by pruning function calls from the dependency graph; guiding the developer to add function calls by detecting missing labels and querying a factbase of functions that add these labels to data; and removing misplaced labels from types or functions. The error message in Listing 1 shows two repair suggestions that remove or insert function calls.

4 Prototype

Paralegal is a prototype privacy linter that follows the design in §3. It consists of 6,370 lines of Rust and uses the Flowistry analyzer [8] and Alloy [13], a flexible layering of first-order logic atop a SAT-solver that serves as both a bounded model checker and as a general solver.

The developer picks the starting point(s) of the analysis by annotating a function with the `analyze` annotation. This allows developers to identify parts of their code that are most relevant to the analysis; for example, in a web application, it is more meaningful to reason about individual controllers as units of user-interaction than analyzing a `main` function. Privacy properties can also reason across these analysis entry-points. The analysis recurses into functions for which Paralegal has access to the source code and which are not explicitly described by the developer via labels. Other functions are approximated using their type signature.

Limitations. The privacy linter works with an abstracted version of the program, which necessarily sacrifices fidelity

relative to the code. The granularity of the system model is lifted to the level of callsites, which means that source code details apart from function calls are reduced to data- and control-flow dependencies. Functions might also be insufficiently approximated, as the linter models their behavior according to their Rust type signature. These semantic gaps can cause the privacy linter to both miss violations that exist and report spurious ones, as our evaluation will show.

5 Evaluation

We applied Paralegal to Websubmit, a homework submission system deployed at a U.S. university and written in approximately 1,600 lines of Rust. We evaluated whether Paralegal detects violations in two application endpoints at different annotation efforts, and we assessed Paralegal's repair suggestions.

We tested three privacy properties: *data deletion* (§2); *scoped storage*, a precondition for deletion that requires personal data to be stored alongside a user identifier; and *authorized disclosure*, which encodes the application's access control policy. Authorized disclosure allows students to see their own answers and answers for classes that they lead as well as allowing instructors to see all answers, including course feedback.

To evaluate annotation and property writing effort, we tested three setups, each with different sets of labels and privacy properties.

(1) *Library*: Adds 16 labels to library code, but none to application code. This represents a one-time effort performed by library authors. Privacy policies are encoded in a generic way around the library labels.
(2) *Application*: The developer adds 20 labels to Websubmit's code. These labels help refine the privacy properties to reason about application-specific semantics such as distinguishing students from instructors.
(3) *Strict*: Adds no further labels, but includes semantic information about vectors and approximations for loops in the environment model. This is additional work for the privacy engineer and ties the properties more closely to the implementation.

Websubmit out-of-the-box complies with the three properties and passes the linter in all setups. To evaluate Paralegal's ability to find issues, we identified "articulation points": points in each privacy property that a program might fail to satisfy. We then translated these articulation points into edits to Websubmit's code, essentially creating various incompliant versions of the program. For example, one articulation point for data deletion is the part of the property that says "of every type." One error for this articulation point is forgetting to delete one type of personal data, such as in (§2). We ultimately evaluated three edits for data deletion, one for scoped storage, and three for authorized disclosure.

Edit Type	Property	Linter Errors Found		
		Library	Application	Strict
Bug	Deletion	2/3	2/3	3/3
	Storage	0/1	0/1	1/1
	Disclosure	0/3	3/3	3/3
Intentional	Deletion	1/3	1/3	2/3
	Storage	1/1	1/1	1/1
	Disclosure	2/3	3/3	3/3

(a) *Number of errors found / total errors.*

Edit Type	Property	Spurious Errors Found		
		Library	Application	Strict
Alternative	Deletion	1/3	1/3	1/3
	Storage	0/1	0/1	0/1
	Disclosure	1/3	1/3	1/3

(b) *Number of spurious errors / "alternative" implementations.*

Figure 3: *Good, acceptable and bad results of testing different code versions. Paralegal finds most errors and reports few spurious ones, and higher annotation effort (in the Application and Strict setups) translates into better performance.*

Errors Caught. For each articulation point, we made two kinds of error-inducing edits. "Bug" edits correspond to a typical, unintentional programmer mistake, such as the above example. By contrast, "intentional" edits reflect a developer intentionally trying to circumvent the linter. An example of this for the data deletion property would be deleting dummy data instead of actual user data just to satisfy the property's requirement that there be *some* invocation of a delete method on a given type.

A good result would show that i) Paralegal detects both categories of errors and ii) the number of detected errors increases as developers annotate more code and property writers write more specific privacy properties in the *Application* and *Strict* settings. Figure 3a shows the results. As predicted, the number of errors caught grows with annotation and property engineering effort (left-to-right). Despite the limited amount of information available to Paralegal in the *Library* setup, it finds about a third of the issues. The poor performance for "bug" edits on data disclosure in this setup arises because Paralegal cannot reason about application-specific access control policies when only given properties over generic library annotations. In the *Application* and *Strict* setups, Paralegal finds 10 and 14 out of 15 issues respectively.

Spurious Errors. For a privacy linter to be practical, it should not report many spurious errors. To evaluate this, we create an "alternative" edit at each articulation point that changes the program syntax, but preserves its semantics: for

example, replacing a `for` loop with a `for_each` call on an iterator. Since the edited programs are semantically equivalent to the original program that passes the linter, a good result would show all of them passing the linter.

Figure 3b shows that five of seven edits pass Paralegal, but Paralegal falsely identifies two as error. One of these happens because Flowistry's ability to reason about dependencies within closures is limited; the other occurs because our version of the property specification does not allow for authorization checks via `if` conditionals. The fact that the results are the same across the three setups is encouraging, since a potential downside of making properties more specific in the *Application* and *Strict* is that it may make properties brittle to benign edits to source code.

Repair Suggestions. For bugs in the *Application* setup, Paralegal provides a correct fix for two of the five problems it catches. In two additional cases, Paralegal informs the developer of a missing label and points to where in the program it expects that label. This suggestion guides the developer in the right direction, but still requires them to come up with the actual fix.

6 Related Work

Similar to the way that traditional linters check for "code smells," related work exists for automatically detecting "privacy smells" [14]; however, these tools function by observing program behavior at runtime. There has also been a progression in code linters (which previously only analyzed abstract syntax trees) to traverse data- and control-flow graphs for additional information [2] to discover a *fixed* set of problematic patterns. Although some linters do allow for users to customize properties to detect, such as through DSLs [16], they are typically based on *syntactic* criteria, and would therefore fare poorly on semantic properties.

There is also a large body of prior related research on information flow control (IFC). IFC systems offer *guarantees* about non-interference, covering certain important privacy concerns, but not others such as data deletion. However, it comes at a high buy-in cost. Existing static IFC frameworks require special languages [6] or extend existing ones with extensive annotations that go beyond functions and types directly relevant for privacy [5, 7, 17, 27]. Nevertheless, the powerful reasoning capabilities of IFC inspired our ideas for system analysis and repair [19, 20]. PrivGuard [29] is an exception, as it works on Python code, but its abstract analysis is limited to small programs due to pointer semantics, and all library functions must be modeled explicitly. Grok [22], by contrast, achieves low developer burden by targeting MapReduce as a restricted data flow language. Dynamic IFC requires specialized runtimes and leads to unpredictable behaviors at runtime, either crashing applications [18] or substituting default values [28, 31].

Lastly, model checking has previously been successfully used to verify privacy related concerns such as access control [9, 11, 12, 21, 32], security policies [15], and preventing data leakage [4]. We build on their techniques to check general privacy properties directly on source code with low manual modeling effort.

7 Open Questions

We have presented the design, and proof-of-concept, of a privacy linter. We build atop powerful formal tools such as program analysis, model checking, and ownership types. Yet, like other linters, instead of genuflecting at the altars of formal properties (like soundness), we stress utility and low false-positives, and embrace ergonomic concerns like incremental adoption. At the same time, we attempt to avoid syntactic brittleness and provide valuable features like suggesting repairs to fix violations. We now discuss what it would go from concept to reality at scale.

How to scale the linter? A strength of classical linters is that their reasoning is local, so they scale well as programs grow. Powerful program analyses can enable global reasoning, which reduces false-positives, but can have trouble scaling. The typical way to ameliorate this cost is to employ modularity through summaries of bodies of code. This also enables incremental re-analysis. It is not yet clear what the right notion of modular interface would be for our setting. However, the approximate nature of linting opens the possibility of sacrificing accuracy for performance.

In our current design, the flow analysis is independent of properties. That means it cannot exploit the properties to summarize what functions do. We believe it might be feasible to create a *property-aware* flow analyzer that produces *sufficient* summaries for verifying uses of that function. This would improve scalability by producing small summaries that nevertheless do not miss potentially problematic code.

Repair. Our design suggests the idea of repairing buggy programs. Repair is amenable to much more sophisticated program synthesis approaches than we use in our prototype. For now, various kinds of synthesis seem to work best on small programs. If our linter is applied frequently during development, then property failure can often be localized to recently-modified code, which may enable the effective use of synthesis.

From policies to properties. Policies originate outside the technical realm (for instance, from laws like GDPR). We have glossed over the important step of translating them into properties that the linter can check. This requires mapping the vocabulary of the policy to the program's semantics (by appropriately labeling data and control elements), and using those labels to translate from the legal code to the property language. Ultimately, these mechanisms must enable people

whose expertise lies outside the technical sphere to be able to author and reason about these properties.

Complementing the toolkit. We presented our design in terms of static analyses and static enforcement. On the analytic side, a linter can incorporate other tools, like symbolic evaluators, to improve precision where needed, especially around potential violations. On the enforcement side, there is a venerable tradition [30] of complementing static checks with program monitoring to complete part of the enforcement at run-time. The latter idea affects the cost of program execution (and may have other subtle impacts in languages like Rust), so they would need to be applied with care and on programmer demand.

References

[1] *AdultFriendFinder Network Hack Exposes 412 Million Accounts*. URL: https://www.zdnet.com/article/adultfriendfinder-network-hack-exposes-secrets-of-412-million-users/ (visited on 01/23/2023).

[2] Nabil Almashfi and Lunjin Lu. "Code Smell Detection Tool for Java Script Programs". In: *2020 5th International Conference on Computer and Communication Systems (ICCCS)*. 2020, pages 172–176.

[3] Anonymized. *Websubmit-Rs: A Simple Class Submission System*. Oct. 2021. URL: Anonymized (visited on 04/06/2022).

[4] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. "Towards Model Checking Android Applications". In: *IEEE Transactions on Software Engineering* 44.6 (June 2018), pages 595–612.

[5] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. "HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. New York, NY, USA: Association for Computing Machinery, Aug. 2015, pages 289–301.

[6] Adam Chlipala. "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. USA: USENIX Association, Oct. 2010, pages 105–118.

[7] Stephen Chong, K. Vikram, and Andrew C. Myers. "SIF: Enforcing Confidentiality and Integrity in Web Applications". In: *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. SS'07. USA: USENIX Association, Aug. 2007, pages 1–16.

[8] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. "Modular Information Flow through Ownership". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. New York, NY, USA: Association for Computing Machinery, June 2022, pages 1–14.

[9] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. "Specifying and Reasoning About Dynamic Access-Control Policies". In: *Automated Reasoning*. Edited by Ulrich Furbach and Natarajan Shankar. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pages 632–646.

[10] Ming Fan, Le Yu, Sen Chen, Hao Zhou, Xiapu Luo, Shuyue Li, Yang Liu, Jun Liu, and Ting Liu. "An Empirical Evaluation of GDPR Compliance Violations in Android mHealth Apps". In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. Oct. 2020, pages 253–264.

[11] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. "Verification and Change-Impact Analysis of Access-Control Policies". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. New York, NY, USA: Association for Computing Machinery, May 2005, pages 196–205.

[12] Dimitar P. Guelev, Mark Ryan, and Pierre Yves Schobbens. "Model-Checking Access Control Policies". In: *Information Security*. Edited by Kan Zhang and Yuliang Zheng. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pages 219–230.

[13] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation". In: *ACM Transactions on Software Engineering and Methodology* 11.2 (Apr. 2002), pages 256–290.

[14] Immanuel Kunz, Angelika Schneider, and Christian Banse. "Privacy Smells: Detecting Privacy Problems in Cloud Architectures". In: *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2020, pages 1324–1331.

[15] Jianli Ma, Dongfang Zhang, Guoai Xu, and Yixian Yang. "Model Checking Based Security Policy Verification and Validation". In: *2010 2nd International Workshop on Intelligent Systems and Applications*. May 2010, pages 1–4.

[16] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Transactions on Software Engineering* 36.1 (2010), pages 20–36.

[17] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. New York, NY,

USA: Association for Computing Machinery, Jan. 1999, pages 228–241.

[18] James Parker, Niki Vazou, and Michael Hicks. "LWeb: Information Flow Security for Multi-Tier Web Applications". In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), 75:1–75:30.

[19] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. "Liquid Information Flow Control". In: *Proceedings of the ACM on Programming Languages* 4.ICFP (Aug. 2020), 105:1–105:30.

[20] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. "Type-Driven Repair for Information Flow Security". In: *CoRR* (2016).

[21] Amit Sasturkar, Ping Yang, Scott D. Stoller, and C. R. Ramakrishnan. "Policy Analysis for Administrative Role-Based Access Control". In: *Theoretical Computer Science* 412.44 (Oct. 2011), pages 6208–6234.

[22] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. "Bootstrapping Privacy Compliance in Big Data Systems". In: *2014 IEEE Symposium on Security and Privacy*. May 2014, pages 327–342.

[23] Oliver Smith. *The GDPR Racket: Who's Making Money From This $9bn Business Shakedown*. URL: https://www.forbes.com/sites/oliversmith/2018/05/02/the-gdpr-racket-whos-making-money-from-this-9bn-business-shakedown/ (visited on 02/01/2023).

[24] *Spain: AEPD Fines El Periódico de Catalunya €10,000 for Violation of Right to Erasure*. July 2020. URL: https://www.dataguidance.com/news/spain-aepd-fines-el-peri%C3%B3dico-de-catalunya-10000 (visited on 01/25/2023).

[25] *Spanish DPA Fines Vodafone Spain More than 8 Million Euros | European Data Protection Board*. URL: https://edpb.europa.eu/news/national-news/2021/spanish-dpa-fines-vodafone-spain-more-8-million-euros_en (visited on 01/25/2023).

[26] *The Cost of Continuous Compliance*. Feb. 2020. URL: https://www.datagrail.io/resources/reports/gdpr-ccpa-cost-report/ (visited on 02/01/2023).

[27] Shukun Tokas, Olaf Owe, and Toktam Ramezanifarkhani. "Static Checking of GDPR-related Privacy Compliance for Object-Oriented Distributed Systems". In: *Journal of Logical and Algebraic Methods in Programming* 125 (Feb. 2022), page 100733.

[28] Frank Wang, Ronny Ko, and James Mickens. "Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pages 615–630.

[29] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. "PrivGuard: Privacy Regulation Compliance Made Easier". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pages 3753–3770.

[30] Andrew K. Wright and Robert Cartwright. "A Practical Soft Type System for Scheme". In: *SIGPLAN Lisp Pointers* VII.3 (July 1994), pages 250–262.

[31] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, Dynamic Information Flow for Database-Backed Applications". In: *ACM SIGPLAN Notices* 51.6 (June 2016), pages 631–647.

[32] Nan Zhang, Mark Ryan, and Dimitar P. Guelev. "Evaluating Access Control Policies Through Model Checking". In: *Information Security*. Edited by Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pages 446–460.

[33] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven Bellovin, and Joel Reidenberg. "Automated Analysis of Privacy Requirements for Mobile Apps". In: *2016 AAAI Fall Symposium Series*. Sept. 2016.