MagnetDroid: security-oriented analysis for bridging privacy and law for Android applications

Emantiele Uliana and Kostas Stathis  
Department of Computer Science  
Royal Holloway University of London  
UK  
Emanuele.Uliana.2016,Kostas.Stathis@rhul.ac.uk

Robert Jago  
School of Law  
Royal Holloway University of London  
UK  
robert.jago@rhul.ac.uk

ABSTRACT
MagnetDroid is a novel artificial intelligence framework that integrates a security ontology, a multi-agent organisation, and a logical reasoning procedure to help build a bridge between the worlds of Android application analysis and law, with respect to privacy. Our contribution helps identify violations of the law by Android applications, as well as predict legal consequences. The resulting implementation of MagnetDroid can be useful to privacy-concerned users in order to acknowledge problems with the privacy of the applications they use, to application developers/publishers to help them identify which problems to fix, and to lawyers in order to provide an additional level of interpretation for any court when considering the privacy of Android applications.

CCS CONCEPTS
- Security and privacy; - Computing methodologies → Multi-agent systems; - Applied computing → Law;

KEYWORDS

ACM Reference Format:

1 INTRODUCTION
In our modern data-driven world, most, if not all the information regarding a specific individual or group can be recorded through data. Therefore, data must assume a central role when characterising the concept of privacy, not in an abstract philosophical sense, but from a modern and pragmatic point of view. It is not far-fetched to say that today privacy of individuals is asymptotic to the privacy of their data. As data, by their own nature, are made to be stored, processed, and exchanged, it is natural to identify security as a necessary condition for their privacy. In particular, no data can be (remain) private if the storage system, the processing routines, and the exchanging infrastructure(s) are not secure enough with respect to the state-of-the-art of known attacks.

While this is true in general, we choose to focus our attention on a specific subset of the global data environment, namely, mobile operating systems (OSs), and, more specifically, Android applications. Android, by far the most popular mobile OS[2], comes with its own security model[4]. However, such security model, while lowering some of the risks identifiable in a mobile OS threat model, leaves significant security decisions in the hands of developers, users, and third parties (e.g., servers applications exchange data with). In practice, this contributes to the existence of an entire ecosystem of vulnerable applications which expose the data of their users to higher risks;

malicious applications whose only purpose is to trick their users or perform activities behind their back, enhancing the probability of misuse of private data.

Society has responded to the issue of vulnerable/malicious applications in the same way it managed vulnerable and malicious software for non-mobile operating system. From the technological point of view, vendors patch vulnerabilities, and third parties develop analysis tools to be able to identify vulnerabilities, and malicious behaviours.

From the legal point of view, countries have introduced rules and regulations on data protection (e.g. in the UK the Data Protection Act 2018 (which implements the GDPR[17]), and The Network and Information Systems Regulations 2018 which aim to discourage carelessness (of developers) and maliciousness (of attackers), and to punish them, when applicable, in case of incidents.

The general problem with those approaches is that they hardly inter-operate, and sometimes attempt to pursue conflicting goals, such as in the long standing conflict between end-to-end encryption and state-mandated backdoors in cryptographic primitives [21]. The problem can be further exemplified by the fact that, usually, Android application analysis tools do not care about (factor in) the law. One reason is that the security of an application does not depend on the law. Another reason lies in the fact that the developers of those tools, usually, are not that knowledgeable about the legal position. Also, the law can often reference vague technological concepts (e.g., confidentiality) without defining them, leaving the door open to (sometimes conflicting) interpretations by different courts. This uncertainty, which is already a potential problem for the developers of Android applications, does not make the legal issues attractive for developers of analysis tools either.
The goal of our work is to contribute towards bridging the gap between technology and law by providing a technology-guided interpretation of a subset of the Data Protection Act 2018 and The Network and Information Systems Regulations 2018 in the context of Android applications. The contribution of our work is that we identify necessary conditions for the gap to be bridged, i.e., unity on the technological side, and a model for the law. In particular, we propose an Android Security Ontology (ASO), and a procedure to aggregate heterogeneous reports from different Android application analysis tools under a common syntax/semantics which enables logical reasoning. We also propose a multi-agent platform to perform the aforementioned procedure. We propose a model of a subset of the Data Protection Act 2018 and The Network and Information Systems Regulations 2018 which enables logical reasoning as well. We explore logical reasoning techniques in order to develop a knowledge-based procedure which, given the previous results, is able to detect incompatibilities of Android applications with what the law prescribes in terms of security, and attempts to foresee the legal consequences of violations and incidents.

In Section 2 we discuss the background surrounding our work, as well as the state-of-the-art and its limitations. In Section 3.1 we describe the Android Security Ontology. In Section 3.2 we describe the flow to generate a technological knowledge base from an Android application and a set of application analysis tools. In Section 4 we describe our model (i.e., legal knowledge base) of the Data Protection Act 2018 and The Network and Information Systems Regulations 2018. In Section 4.2 we describe how to use the technological knowledge base and the legal knowledge base to derive incompatibilities and predict legal consequences. In Section 5 we show an example of our work in a practical scenario. In Section 6 we explain the limitations of our work, together with planned directions for future work.

2 BACKGROUND AND EXISTING WORK

Bridging the worlds of technology and law in the context of Android requires an understanding of how each view manages the problems of vulnerable and malicious applications. Ideally, we would like to gather the results on the technological side, and use an aggregation of them as a technological knowledge base. Security researchers have ported the same strategies that proved successful with non-mobile software. In particular, they have extended the two main approaches (static and dynamic analysis) to work with mobile applications. The static approach leverages (source or machine) code analysis of a software program to discover behavioural patterns, while the dynamic approach executes the program in a (real or emulated) monitored environment in order to trigger and record notable behaviours.

Related works on Android application analysis. Among the relevant work on application analysis of Android applications, we can cite Babelview[31], Androguard[14], MalloDroid[18], FlowDroid[6], Apposcapy[19], and Dexpler[7] as examples of static analysis. On the other hand, DroidScope[45], CopperDroid[38], PuppetDroid[20], TaintDroid[16], DroidTrace[47], Andlantis[10], and IntelliDroid[44] mainly employ dynamic analysis. There have been attempts to bridge the two families of approaches, such as SMV-HUNTER[35], Andrubis[25], and Andrototal[26]. Static and dynamic analysis have their own pros and cons[15], and some of them are peculiar and unique to mobile OSs.[15] However, the number of unique Android applications is raising at a fast pace[2], and a certain degree of automation is needed in order to be able to analyse the raw amount of applications in a reasonable time. Here we encounter the first problem: usually the analysis tools are standalone, and their reports have arbitrary syntax and semantics. In other words, there is no underlying common ontology. The reports are mainly meant for humans to read, as natural language is mainly featured. For this reason, they are not easily understood by automated software programs. It is hard and demanding to automatically aggregate different reports from different tools, in order to merge the information they provide. This is a problem for us, because, if we want a technological knowledge base comprising a significant variety of security issues, we have to aggregate reports deriving from the analysis of a certain application with different tools. Aggregation of heterogeneous sources of information has been attempted in the past by means of leveraging agent platforms, as discussed by Ishii et al [22]. As our problem is very specific, we cannot reuse existing agent frameworks. However, we agree that a multi-agent platform is a suitable component for our needs.

Related works on ontologies for Android. On the subject of specifying ontologies in the context of Android, we can cite: Android goes Semantic: DL Reasoners on Smartphones[46], A Power Consumption Benchmark Framework for Ontology Reasoning on Android Devices[40], A linguistic mobile decision support system based on fuzzy ontology to facilitate knowledge mobilization[28], A user profile ontology based approach for assisting people with dementia in mobile environments[34], A smart indoor navigation solution based on building information model and google android[33], Android Based Effective and Efficient Search Engine Retrieval System Using Ontology[24], and Privacy protection for smartphones: an ontology-based firewall[41]. Unfortunately, none of those captures the relevant security concepts we expect to find inside the reports generated by application analysis tools. In light of this, one of our contributions is to create a custom Android Security Ontology (ASO) which then can be used as a guide to aggregate heterogeneous reports.

Related legal background. In legal terms, the provisions regarding data protection (e.g., the Data Protection Act 2018 [1], and The Network and Information Systems Regulations 2018 [3]) are relevant to our needs. We can identify a twofold approach to data protection. On one side, we have law with a prescriptive function (LPF, or prescriptive law from now on) which mainly enumerates obligations for parties (e.g., developers) with respect to some properties (e.g., data confidentiality, data integrity, data authenticity) which are left open to interpretation. On the other side, we have law with a consequence function (LCF, or consequential law from now on) which details the triggered consequences should certain events (e.g., data leaks) happen. It is important to note that the consequences only apply after the incident has happened, not if there is the potential (i.e., all the conditions are met) for the incident to happen. As the law is written in natural language, it is even harder to understand for an automated system. Moreover, since the law is always subject to a certain degree of interpretation by lawyers during court cases, its formulation is not at all automation-friendly. We need to
model the relevant rules and regulations in order to create a legal knowledge base suitable for our purposes.

3 MAGNETDROID FRAMEWORK

In order to address the issues identified in Section 2, we discuss:

- an Android Security Ontology to specify a common syntax/semantics for analysis tools reports;
- a general architecture for retrieving, translating, and aggregating reports in the form of a multi-agent platform;
- a model of a subset of the Data Protection Act 2018 and The Network and Information Systems Regulations 2018;
- a reasoning procedure to use the results of the previous points to discover incompatibilities between the analysed application(s) and the LPF, as well as predict legal consequences derived from the LCF.

3.1 Android Security Ontology (ASO)

As anticipated in Section 2, we would like to re-use existing Android application analysis tools in order to produce reports of security problems. However, running analysis tools on a specific application usually results in a collection of heterogeneous documents which are hard to automatically post-process. For this reason, our ASO\(^1\) guides the translation and aggregation procedure which produces a final report wrapping a so called technological knowledge base.

Our ASO is a tree-like structure where each node is a concept related to the security of Android applications, and each parent-child relation is either a partition of the parent into its components (e.g., permissions into protection_normal, protection_signature, dangerous, spacial), or a link from a more general concept (e.g., network_activity) to a more specific one (e.g., observed_protocols). The root is android_application, and its immediate children are assets, risks, threats, vulnerabilities, behaviours, and features. The leaves consist of fine-grained concepts that we expect to find in reports such as broken_tls, meaning the use of insecure parameters with the network communication protocol TLS [5]. The structure of internal and leaf nodes is shown in Figure 1. Each leaf may contain one or more state variables. Each state variable is a container for structured information on a very specific aspect of the security of Android applications, plus some custom indexes and classes which help characterising the severity of the problem (if any), and its impact under two different mindsets:

- **credulous**: an optimistic view such that problems are not really considered unless they have an immediate and catastrophic effect on the entire user-app-provider environment.
- **skeptical**: a pessimistic view that treats every discovered problem/vulnerability as if it intrinsically had an immediate and catastrophic effect on such environment.

The motivation for the multiple mindsets lies in one of the practical uses for ASO, namely the creation of a technological knowledge base (described in Section 3.2) which enables logical reasoning (described in Section 4) together with a legal knowledge base (also described in Section 4). In particular, it transcends the bounds of technology, as it is rooted in the concept of interpretation of the law. Depending on the interpretation, problems may be considered relevant:

- only if even the most optimistic (i.e., credulous) view recognises them as important;
- as soon as the most pessimistic (i.e., sceptical) view recognises them as important.

Therefore, we capture this twofold interpretation by separating our mindset into two. In order to measure each mindset numerically, we make use of two pairs of index-class. In particular, the credulous approach is represented by the minimal impact index, and the minimal impact class, while the sceptical approach is represented by the maximal impact index, and the maximal impact class. The meaning of those indexes, and their usage is detailed in Section 3.2, and Section 4.

The absolute relevance index, and the absolute relevance class, on the other hand, represent how much weight (i.e., consider important/relevant) a state variable carries in absolute terms. The absolute index and class are an intrinsic property of the state variable, and, as such do not change depending on the current_state value of the variable. The structure of state variables is depicted in Figure 1.

The importance of the structure of ASO is that it offers us a guide for translating tool reports whose format is often arbitrary. In this way, we can use ASO as a common and systematic format for reports, which is currently lacking. We will see later in Section 3.2, that the translation of a report consists of an instantiation of ASO, by means of selecting a meaningful value for every field (except the static ones e.g., id, name, possible_states, and the absolute indexes and classes) of each state variable. In addition, the value to assign

---

\(^1\)For a detailed demo see https://dicelab.co.uk/aso.html.

Figure 1: A diagram illustrating the internal structure of nodes, leaf nodes, and state variables.
to each field of a particular state variable is selected in accordance with the content(s) of the report to translate.

It is possible that the report does not cover the subject a state variable refers to. In this case, ASO provides the reserved value UNASSIGNED as a backup strategy. Also, ASO supports an additional reserved value named CONFLICTING. We will see later in Section 3.2 that this value is mainly useful during the aggregation of translated documents, in case an unsolvable conflict is found. Its internal structure has been crafted so to keep track of the disagreement in case a solution is found in the future (e.g., by means of a manual validation). The aforementioned internal structure, as well as the trivial structure of UNASSIGNED are shown in Figure 1.

3.2 MagnetDroid: creating the technological knowledge base

The ASO can be used as a guide to translate and aggregate reports from Android application analysis tools. We present a multi-agent framework to produce and collect raw reports from available analysis tools, perform the translation and aggregate the translated reports to produce a so called final report to wrap our technological knowledge base. Our agent architecture, inspired by GOLEM [11] consists of an environment containing a pre-specified set of agents, each assigned with a specific role to play in the system (see Section 3.2.1). Our agents are characterised by a body that wraps a set of sensors for the agent to perceive the environment, a set of actuators for the agent to change the environment, and a mind for the agent to decide which action(s) to execute next. The mind is characterised by reasoning procedures (as in the architecture proposed by the KGP model of agency [23]), so that it is able think which action is best to perform, when confronted with multiple alternatives. In this context, the agent mind is in a perpetual cycle whose steps are

(1) perceive(), which fetches any new perception available from the sensors;
(2) revise(), which revises the internal state and beliefs of the mind, when new sensing information is received;
(3) decide(), which determines the next action to execute in the environment;
(4) execute(), which triggers the execution by propagating to the body the action decided.

Each agents has its own implementation of the four steps, and it own pool of available actions (see Section 3.2.1). In the current version of MagnetDroid, the decision process follows the teleo-reactive programming paradigm [29, 32], which allows the agent to exhibit behaviours defined as a set of condition-action rules, each set being indexed by the implicit goal that the behaviour achieves. Such behaviours have the form:

\[ G = \{ C_1 \rightarrow A_1; C_2 \rightarrow A_2; \ldots; C_n \rightarrow A_n \} \]

where \( G \) is the goal that the behaviour achieves, \( C_i \) with \( 1 \leq i \leq n \) being the conditions of the rules, while \( A_i \) are the actions that need to executed. At each cycle step, one of these rules succeeds, only if the conditions are satisfied in the internal state of the agent, and then the action is executed. The listing below shows the cycle of a WorkerAgent (discussed in Section 3.2.1) and illustrates how the top-level decide() is formulated as a teleo-reactive behaviour:

```java
public void perceive() {
    fetchPercepts(); stream(); forEach(this::storePercepts);
}

public void revise() {
    if(this.isAnalysisStarted) {
        this.isAnalysisStarted = checkForAnalysisStarted();
    }
    if(this.isAnalysisStarted && this.isAnalysisFinished) {
        this.isAnalysisFinished = checkForAnalysisFinished();
    }
    else if(this.isAnalysisFinished && !this.isReportReady) {
        this.reportReady = checkForReportReady();
    }
}

public MagnetDroidAction decide() {
    if(this.isReportReady) {
        return new SendReportToCoordinatorAction(this.report);
    }
    else if(this.isAnalysisFinished) {
        return new RetrieveReportAction();
    }
    else if(this.isAnalysisStarted) {
        return new StartAnalysisAction(this.apk);
    }
    else {
        return manageToolExecution();
    }
}

public void execute(MagnetDroidAction action) {
    sendToBody(action);
}
```

3.2.1 The flow of the phases. In order to use our multi-agent framework to translate and aggregate reports from Android application analysis tools, we designed a flow consisting of 3 distinct phases:

(1) Phase 1: parallel analysis.
(2) Phase 2: translation.
(3) Phase 3: aggregation.

Figure 2 depicts a schematic visualisation of the phases.

![Figure 2: A diagram of the phases from the APK and the tools to the final report.](image-url)
Collecting the reports. In Phase 1: parallel analysis, we have a CoordinatorAgent which is given an APK (essentially a zip file containing the Android application), and the knowledge of the set of available analysis tools. The CoordinatorAgent dispatches a set of WorkerAgents, one for each available tool, in order to run each tool on the APK, provide the necessary inputs (if applicable), and retrieve the generated report.

Parsing the raw reports. Phase 2: translation is divided into sub-phases: parsing and ASO instantiation. During the first half, a highly specialised ParsingAgent receives a report from the CoordinatorAgent. Immediately, it attempts to classify it as either a narrative or factual report. Narrative refers to the fact that time is featured and important in the report. In particular, the report itself is characterised by a series of events with either explicit, or implicit (i.e., the ordering) timestamps. Factual, on the other side, means that time is not featured and that it is irrelevant, as the report is essentially a collection of facts about the analysed application. More often than not, the class of a report depends exclusively on the tool used to produce it. Therefore, the ParsingAgent guesses the class based on the name of the tool. Then, the ParsingAgent polishes the report by removing unnecessary information that is useless for the translation (e.g., welcome messages and unreasonably verbose debug messages). This step varies in complexity depending on the nature of the report (narrative vs. factual), and ultimately, on the tool which produces the report in the first place. Section 5 shows an example of what a narrative and a factual report look like. The ParsingAgent abstracts from narrative reports an ordered sequence of event occurrences, which we represent here in a logic-based format of Prolog (where capitalised names of arguments denote variables and names starting with lower case letters denote constants):

\[
\begin{align*}
\text{happens\_at}(\text{Timestamp}_i, \text{Event}_i) \\
\text{happens\_at}(\text{Timestamp}_N, \text{Event}_N)
\end{align*}
\]

where \( \text{Timestamp}_i \) orders the occurrence of the description contained in an \( \text{Event}_i \) which in turn contains information about the content of a report (and \( 1 \leq i \leq N \)). Likewise, the ParsingAgent abstracts from factual reports a non-ordered collection of facts represented as

\[
\begin{align*}
\text{holds\_in}(\text{ReportId}, \text{Fact}_i) \\
\text{...}
\end{align*}
\]

where \( \text{ReportId} \) is the identifier of the report and \( \text{Fact}_i \) is a piece of relevant information about the application and/or its behaviour during the analysis phase.

Instantiating the ASO. When the ParsingAgent has completed its task, it passes the polished report to a highly-specialised TranslatingAgent which is responsible for the second half on the translation, namely instantiating ASO. The TranslatingAgent has a perfect knowledge of ASO, and is able to interpret the contents of a polished report produced by a specific ParsingAgent. The TranslatingAgent iterates through every leaf of ASO, and, for every leaf, for every state variable, checks within the polished report for useful information in order to assign a value to the relevant fields of the state variable. In case such information does not exist within the polished report, the special value \text{UNASSIGNED} is assigned instead.

Aggregating the reports. After all the reports have been translated, the TranslatingAgents report back to the CoordinatorAgent, which now dispatches an AggregatingAgent with the task to aggregate the translated reports, initiating Phase 3: aggregation. Since the translated reports are nothing else than instantiations of ASO, they share common syntax and semantics, and, therefore, it is much easier to define an automated procedure to aggregate the information contained in each of them. The aggregation procedure loops through the branches of ASO, and, for every translated report, fetches the state variables, and tries to merge them. During such process, a conflict arises when, for a specific field of a specific state variable, different translated reports disagree on the value, or one of the values is, \text{CONFLICTING} itself. If there is no reasonable way to derive an agreed value from the conflicting values, we have an unsolvable conflict. Otherwise, if a value compatible with all the disagreeing values can be derived, that becomes the agreed value, and the conflict vanishes.

The result of the aggregation is the final report, which contains our technological knowledge base. In practice, it is an instantiation of ASO containing all the relevant information from the original raw reports combined.

3.2.2 The technological knowledge base. From the final report we are able to explicitly render the technological knowledge base by means of translating state variables into statements expressed in the logic-based language Prolog. In particular, statements are generated to keep track of the parent-child relations, and to bind state variables to leaves.

In case the actual value of a state variable is \text{UNASSIGNED}, the variable is not translated to Prolog. If, however, reports provide conflicting values for the same state variable, the label \text{CONFLICTING} is used for the value, and the variable is included in the final report purely to avoid data loss, since further analysis in the future may lead to a solution of the conflict.

On top of assertions representing instances of state variables in the ASO tree-hierarchy, the technological knowledge base comprises rules which link state variables to the high-level properties of data confidentiality, data integrity, and data authenticity. These rules represent what we call violations. The following listing shows a practical example of a violation rule which, in case of success, indicates that the application uniquely identified by App uses an insecure protocol with notable implications [27] on the privacy of the transmitted data:

\[
\begin{align*}
\text{violated(data\_confidentiality(App, Report), skeptical, rule2)} & : - \\
\text{\_}_{\text{as\_root}(\text{Report, RootID}, ...)} & , \\
\text{\_}_{\text{child\_of}(\text{Report, MetadataID, RootID})} & , \\
\text{\_}_{\text{ancestor\_of}(\text{Report, MetadataID, RootID})} & , \\
\text{\_}_{\text{state\_var}(\text{Report, AppHashID, MetadataID, appHash, App, _, _, _, _, _, _, _})} & , \\
\text{\_}_{\text{ancestor\_of}(\text{Report, TlsID, RootID})} & , \\
\text{\_}_{\text{state\_var}(\text{Report, SSLID, TlsID, "s\_s\_3.0", observed, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _}, \\
\text{\_}_{\text{Mxii}, _}, _)} & , \\
\text{\_}_{\text{Mxii} > 5} & . \\
\text{violated(data\_confidentiality(App, Report), credulous, rule2)} & : - \\
\text{\_}_{\text{as\_root}(\text{Report, RootID}, ...)} & , \\
\text{\_}_{\text{child\_of}(\text{Report, MetadataID, RootID})} & , \\
\text{\_}_{\text{ancestor\_of}(\text{Report, MetadataID, RootID})} & , \\
\text{\_}_{\text{state\_var}(\text{Report, AppHashID, MetadataID, appHash, App, _, _, _, _, _, _, _})} & , \\
\text{\_}_{\text{ancestor\_of}(\text{Report, TlsID, RootID})} & , \\
\text{\_}_{\text{state\_var}(\text{Report, SSLID, TlsID, "s\_s\_3.0", observed, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _}, \\
\text{\_}_{\text{Mxii}, _}, _)} & , \\
\text{\_}_{\text{Mxii} > 9} & .
\end{align*}
\]
Each of the two violation rules above detect the use of an insecure protocol by exploring the ASO tree from the root to the leaf node related to TLS, and then looking at the value of the state variable related to SSL3.0. If such value is observed, SSL3.0 has been observed at least once by at least one tool during Phase 1: parallel analysis.

The two rules differ by means of the approach they take to interpret the final report. In particular, the first rule adopts the skeptical mindset, where even with an average (i.e. 5) maximal impact index (Mxi) the violation is triggered. The second rule, on the other hand, adopts the credulous mindset where, unless there is a high value (i.e. 10) for the minimal impact index (Mii) the violation is not triggered. The labels skeptical and credulous are used in the parameters of the rules, as well as rule labels rule1 and rule2 to uniquely identify such rules. The rule labels are then associated with their purpose, in this case the detection of an insecure protocol under the different mindsets, which can be used as a form of shallow explanation if a violation is detected. The underscore denotes an anonymous variable, meaning that we are not interested in the names of the arguments/parameters in these positions.

We are now in a position to describe if a property of an application related to privacy is satisfied, such as data confidentiality, if all violations in the knowledge base finally fail.

\[
\text{satisfied}(\text{Property}, \text{MindSet}) \iff \neg \text{violated}(\text{Property}, \text{MindSet}, \_).
\]

`\neg` in the rule above is Prolog’s negation, known as negation-as-failure [12]. We can now use the above rule to check that an application is compliant to a number of properties as follows:

\[
\text{compliant}(\text{Properties}, \text{MindSet}) \iff \\
\quad \forall (\text{member}(\text{Property}, \text{Properties}), \text{satisfied}(\text{Property}, \text{MindSet})).
\]

The \text{forall} Prolog predicate will ensure that each member \text{Property} in the list of \text{Properties} that we need to check is satisfied in the specific \text{MindSet}. For example, to check for the privacy properties of an application called ‘a1’ say, under the skeptical mindset, we specify the query:

\[
\text{compliant}([ \text{data\_confidentiality}(a1), \text{data\_integrity}(a1), \text{data\_authenticity}(a1) ], \_). \text{Skeptical}.
\]

The query will use the rules above (and check all the remaining skeptical rules that describe the logical representation of the technological knowledge base) to identify whether these properties are satisfied, and report it to the user.

4 USING MAGNETDROID WITH THE LAW

After modelling the technological side, in order to use our technological knowledge base to construct a bridge between technology and law, we need to build a similar model of the law itself. Across the world, laws are written in natural language, which entails notable problems for any perspective reasoning. In particular, the same sequence of letters (i.e., word) may be used with different meanings (i.e., ambiguity). It is also possible that some concepts are mentioned, but not adequately defined or referenced. Moreover, contradictions between different legislative instruments are a limiting possibility. For these reasons, usually lawyers apply a process of interpretation which can lead to inconsistent results starting from the same premises (i.e., same facts, same rules, different lawyers, different outcomes). As a specification in natural language is a barrier, an abstraction can simplify the problem. Section 4.1 explains our process of abstraction of a model.

4.1 Modelling the law

For simplicity, we only focus on a subset of the Data Protection Act 2018 and The Network and Information Systems Regulations 2018. We are interested in ss66(1) and (2) of chapter 4 of the former, and s1(3)(g) of the latter. Our model first clusters legal rules into 2 different classes: law with prescriptive function, and law with consequential function. The former comprises those legal provisions that specify obligations for parties. The latter comprises those legal provisions that specify consequences for parties, given some other conditions. We create a tree-like structure, similar to ASO with law as the root, and prescriptive and consequential as its immediate children. The parent-child relations have the same range of meanings of those in ASO. The children of prescriptive and consequential are the leaves of the tree.

Prescriptive. Each child of prescriptive represents a specific “prescription” which we identified as relevant for and compatible with the context of privacy and security of Android applications. In particular, it is characterised by a set of Prolog rules whose right sides specify conditions for the truth of the left sides which, in turn, specify obligations.\(^2\) We manually created those rules from the aforementioned articles. For example, we track that a vendor of an application is obliged to secure it for proper use.

Consequential. On the other hand, each child of consequential tracks the potential consequences of events (e.g., incidents such as data leaks) that we identified as relevant for and compatible with the context of privacy and security of Android applications. Much like each consequential leaf, it is characterised by a set of Prolog rules whose right sides specify conditions for the truth of the left sides which, in turn, specify obligations.\(^2\) We manually created those rules from the aforementioned articles. For example, we track that a vendor of an application is bound to hold at some point in the future, therefore allowing us to present the users of our system with a warning type of result in the form of potential consequences.

4.2 The legal knowledge base and post-analysis

Our legal knowledge base consists of a set of Prolog rules derived from our model of the law, both prescriptive and consequential. In the prescriptive case we represent obligations that need to be fulfilled and linked to our technological knowledge base for concepts that are potentially abstract in the law. This means that we will need to introduce technological concepts, for example the notion of mindset, as we have seen in Section 3.2.

In order to exemplify the legal knowledge base, consider the situation where we need to express that a vendor of an application

\(^{2}\) A schematic representation of the model of both the prescriptive and consequential branches is available at https://dicelab.co.uk/law.html.
is obliged to secure it for proper use [1]. We represent this in our framework as:

\[
\text{fulfills} (\text{obliged}(\text{Vendor}, \text{secure}(\text{App})), \text{MindSet}) \leftarrow \text{compliant}(\text{data_confidentiality}(\text{App}), \text{data_integrity}(\text{App}), \text{data_authenticity}(\text{App})).
\]

As mentioned before, we have parameterised the fulfilment of the obligation with the mindset of the compliance checking. The rule is an example of how to link a legal model regarding security to the technological aspects of confidentiality, integrity and authenticity. As those concepts are not conclusively defined in the law, in order to perform reasoning, we need to include the technological knowledge base, so to provide a more comprehensive knowledge base. This is an instance of MagnetDroid acting as a bridge between technology and the law.

In the consequential case, on the other hand, we have condition and consequences, whose relation is captured by Prolog rules. For example, to express that a fine is issued if a data leak is observed, we use the following rule:

\[
\text{issue} (\text{fine}(\text{App}), \text{MindSet}) \leftarrow \text{leak}(\text{App}, \text{MindSet}).
\]

Note that the concept of leak is not explained, nor referenced in the subset of the law we consider. Therefore, once again, we need to refer to the technological knowledge base in order to find a proper definition.

Once we have the union of both knowledge bases, we are able to perform reasoning. In general, we are interested in two classes of findings. First, we want to find obligations that are not fulfilled (vs. the prescriptive law). Second, we want to predict consequences, given specific conditions (i.e., consequential law).

Recalling the fulfill(...) rule from the legal knowledge base -prescriptive, we are now able to perform reasoning, as the technological knowledge base contains rules (see Section 3.2) that allow us to calculate the truth value of data_confidentiality(...), data_integrity(...) and data_authenticity(...) If we find that the truth value for all of the previous is true, then the whole (right side) of the fulfill(...) rule evaluates to true. Consequently, the left side evaluates to true as well, indicating that App fulfils the obligation. Alternatively, if the body of the fulfill(...) rule evaluates to false, then App does not fulfil the obligation.

Recalling the issue(...) rule from the legal knowledge base -consequential, we are now able to perform reasoning, as the technological knowledge base contains rules that allow us to calculate the truth value of leak(...). In particular, one of such rules is represented in the listing below:

\[
\text{leak}(\text{App}, \text{MindSet}) \leftarrow \text{violated} (\text{data_authenticity}(\text{App}, \text{Report}), \text{MindSet}).
\]

If violated(...) evaluates to true, then leak(...) is true as well, which, in turn, causes issue(...) to evaluate to true. In this case, we say that there are the conditions for the issuance of a fine. Alternatively, if violated(...) evaluates to false, the propagation of the truth value causes issue(...) to evaluate to false, meaning that the conditions for the issuance of the fine are not met.

5 CASE STUDY

In order to study the applicability of our proposals, we perform preliminary tests of the flow of the phases (see Section 3.2) on two different APKs. The first is a calculator\(^3\), while the second is a custom-build application which performs some dangerous behaviours, such as sending data via an HTTP POST in plaintext (i.e., unencrypted and to an unauthenticated endpoint). We use Bettercap, AndroTotal and MalloDroid as the available tools. Bettercap, among its functionalities, allows us to intercept the network traffic between two endpoints. AndroTotal provides us with (mainly) signature-based results regarding the maliciousness of an application, according to a pool of anti-malware. MalloDroid searches for misconfigurations within the code of the application with respect to the usage of HTTPS. The following listing shows a snippet of the raw report from Bettercap, edited in order to contain only the meaningful information for the translation. The narrative nature of the report is implied by the timestamp of the event.
Figure 3: A subset of the translated report generated from the original report from Bettercap. Redundant or uninteresting information is either omitted or replaced by ommiss.

Figure 4: A visualisation of the report from AndroTotal.

Figure 5: A subset of the translated report generated from the original report from AndroTotal. Redundant or uninteresting information is either omitted or replaced by ommiss.

Figure 6: A visualisation of the report from AndroTotal.

6 CONCLUSIONS
In our work, we have identified a gap between security-based technology-powered privacy consideration for Android applications and the relevant law. We have built a bridge between these two worlds that analyses these considerations using logical reasoning. Our approach has proposed an Android Security Ontology, a multi-agent platform that translates and aggregates reports from existing Android application analysis tools, and an aggregating procedure that allows us to create what we call a technological knowledge base. We can then use this knowledge base and link it to the relevant legal provision. This allowed us to draw conclusions about the privacy implications of an Android application under a specific interpretation.
We exemplified our approach with a specified model of a subset of the Data Protection Act 2018 and The Network and Information Systems Regulations 2018, in order to create a suitable legal knowledge base. We then performed logical reasoning in Prolog in order to find violations of prescriptive law, and predict legal consequences that may arise from security problems with respect to Android applications. In particular, we chose a calculator application, and a custom-built application which exhibits some of the behaviours we are interested in. We performed queries aimed at detecting the violation of data confidentiality under the skeptical mindset, which, in turn, we used to perform queries aimed at detecting violations of the prescriptive law.

Our work has application in three possible scenarios. The first one involves developers using MagnetDroid to avoid unnecessary vulnerabilities in the applications they build. Here our system can provide shallow explanations of the violations. The second one involves users of applications leveraging MagnetDroid to become aware of the vulnerabilities/maliciousness of the applications they use, along with the legal consequences such violations may have. Finally, the third one involves legal professionals using MagnetDroid to establish technology-based interpretation of facts in specific incidents involving private data of Android users.

MagnetDroid is a proof of concept prototype that integrates a number of complex technologies. Tools from the information security community, distributed agent platforms with a specific agent model and symbolic reasoning techniques based on computational logic. However, in developing the prototype and its use we have identified a number of areas for future work.

- **ASO and its application on specific tools.** Strictly speaking the notion of ontology in ASO as presented in this work is more like an information model because it lacks a formal naming and definition of the categories, properties and relations between the concepts, data and entities that substantiate it. However, we have kept the ontology term here, as we envisage that future implementations of ASO (e.g. in OWL) would make the acronym more accurate. In addition, the translation of raw reports into instantiations of ASO is not provably lossless, and it is currently limited to specific versions of specific tools. As a next step we plan to develop an ASO-compatible API for perspective Android analysis tools developers which would specify a format, the syntax, and the semantics of an ASO-compatible report. Such step would allow us to completely skip the translation of the reports, and to immediately perform the aggregation of the retrieved reports. An ASO-compatible API for tool writers as a future work is inspired by existing similar technologies, e.g., the X.509[13] standard which specifies the format of certificates to be used for authentication of remote parties. It does not matter who the subject the certificate refers to is, or who the issuer is (as long as it is trusted): a software receiving a certificate is able to immediately understand its syntax and semantics without any translation needed.

- **Inconsistencies from tools and interpretation of the law.** Our work in this paper was based on existing Android application analysis tools as the source of the raw reports we translated and aggregated. In doing so, we found that we always inherited the corresponding accuracy, precision, and recall from these tools. One aspect of our framework that mitigated for such cases was the CONFLICTING mechanism. As the conflicting information is likely the result of a mistake by one or more tools, discarding conflicting information allowed us to “discard” the erroneous information, wherever it was. Conflicts may also arise from the open-textured nature of the law, which for the AI and law literature is not a new problem (e.g. [9]). It has been suggested that this type of problem is best dealt with using argumentation-based techniques (e.g. [8]) and a whole area of work has been initiated in this direction as a result (e.g. [30, 42]). In addition, multi-agent platforms that are argumentation-based exist too (e.g. [39]), as well as agent models where argumentation drives an agent’s internal operation (e.g. [43]). Dialogue games are often used in practice to implement these techniques in interactive systems (e.g. [36, 37]).

- **Overfitting to the law.** Our model of the law is derived by means of selecting a subset of a pair of Acts in a single jurisdiction, and performing a manual work of translation from natural language to the tree representation. The creation of the model can be refined in at least two ways: selecting a broader range of rules and regulations, and using natural language processing techniques in order to automate the process of building the model tree.

Dealing with the all of the above issues has opened up important areas that will be the focus of our work for the future.

ACKNOWLEDGMENTS

The authors would like to thank Claudio Rizzo and Roberto Jordaney for inspiring some of the ideas in this work, and Benedict Wilkins and Joel Clarke for commenting on a previous version of the paper. The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. The first author has been supported by a grant from The Magna Carta Doctoral Centre for Individual Freedom at Royal Holloway University of London.

REFERENCES


