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

Emanuele 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

Android, Intelligent Agents, Law, Logic Programming, Ontologies, Privacy, Security.

ACM Reference Format:
Emanuele Uliana and Kostas Stathis and Robert Jago. 2019. MagnetDroid: security-oriented analysis for bridging privacy and law for Android applications. In *Seventeenth International Conference on Artificial Intelligence and Law (ICAIL '19), June 17–21, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3322640.3326729

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.

ICAIL '19, June 17–21, 2019, Montreal, QC, Canada
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6754-7/19/06…$15.00
https://doi.org/10.1145/3322640.3326729

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 preform 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], Apposcopy[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-children 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.

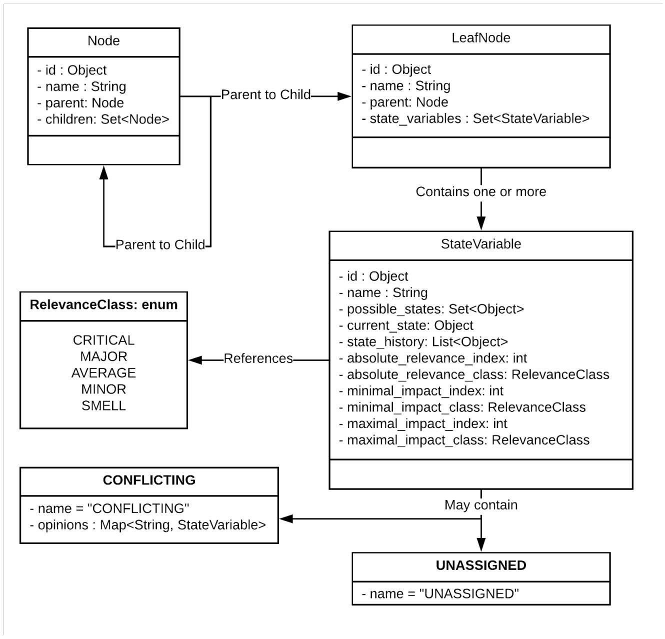


Figure 1: A diagram illustrating the internal structure of nodes, leaf nodes, and state variables.

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., skeptical) 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 *skeptical* 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.

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:

$$G : \{C_1 \rightarrow A_1; C_2 \rightarrow A_2; \cdots ; C_n \rightarrow A_n\}$$

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:

```
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.

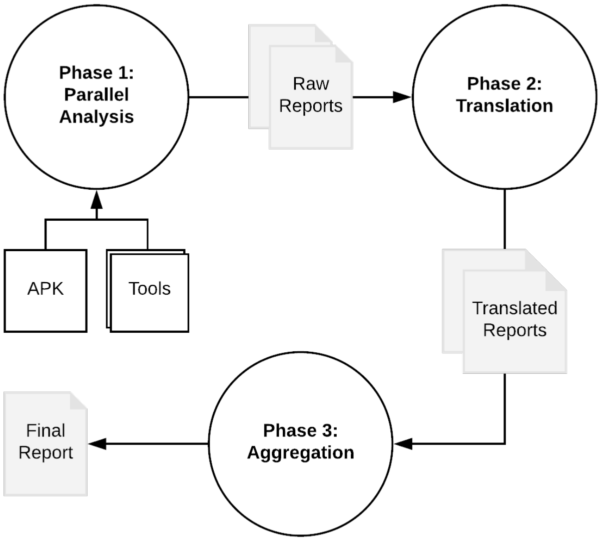


Figure 2: A diagram of the phases from the APK and the tools to the final report.

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 in two 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):

```
happens_at(Timestamp_1, Event1_info).
...
happens_at(Timestamp_N, EventN_info).
```

where $Timestamp_i$ orders the occurrence of the description contained in an $Event_i\_info$, 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

```
holds_in(ReportId, Fact1_info).
...
holds_in(ReportId, FactN_info).
```

where *ReportId* is the identifier of the report and $Fact_i\_info$ 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 *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, *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 *UNASSIGNED*, the variable is not translated to Prolog. If, however, reports provide conflicting values for the same state variable, the label *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:

```
violated(data_confidentiality(App, Report), skeptical, rule1) :-
    aso_root(Report, RootID, _, _, _),
    child_of(Report, MetadataID, RootID),
    ancestor_of(Report, MetadataID, RootID),
    state_var(Report, AppHashID, MetadataID, appHash, App, _, _, _, _, _, _, _),
    ancestor_of(Report, TlsID, RootID),
    state_var(Report, SSLID, TLSID, "ssl3.0", observed, _, _, _, _, _, Mxii, _),
    Mxii > 5.

violated(data_confidentiality(App, Report), credulous, rule2) :-
    aso_root(Report, RootID, _, _, _),
    child_of(Report, MetadataID, RootID),
    ancestor_of(Report, MetadataID, RootID),
    state_var(Report, AppHashID, MetadataID, appHash, App, _, _, _, _, _, _, _),
    ancestor_of(Report, TlsID, RootID),
    state_var(Report, SSLID, TLSID, "ssl3.0", observed, _, _, _, Miii, _, _, _),
    Miii > 9.
```

```
purpose_of(rule1, "Insecure␣SSL3.0␣detected␣under␣the␣skeptical␣approach").

purpose_of(rule2, "Insecure␣SSL3.0␣detected␣under␣the␣credulous␣approach").
```

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 ($Mxii$) 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 ($Miii$) 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 finitely fail.

```
satisfied(Property, MindSet) :- \+ violated(Property, MindSet, _).
```

‘\+’ 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:

```
compliant(Properties, MindSet) :-
    forall(member(Property, Properties), satisfied(Property, MindSet)).
```

The $forall/2$ Prolog predicate will ensure that each member $Property$ in the list of $Properties$ that we need to check is satisfied in the specific $MindSet$. For example, to check for the privacy properties of an application called ’a1’ say, under the *skeptical* mindset, we specify the query:

```
?- compliant([
             data_confidentiality(a1),
             data_integrity(a1),
             data_authenticity(a1)
            ], 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 the 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-children 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, i.e., what must hold for the left sides to become true, while the left sides specify the consequences. It is important to note that, in practice, the consequences only apply after one or more events have taken place, not if there is the possibility of them happening. However, for our purposes of predicting consequences, we treat the conditions as if they are 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:

```
fulfils(obliged(Vendor, secure(App)), MindSet) :-
   compliant([
            data_confidentiality(App),
            data_integrity(App),
            data_authenticity(App)
           ], MindSet).
```

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:

```
issue(fine(App), MindSet) :- leak(App, 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 `fulfils(...)` 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 body (right side) of the `fulfils(...)` 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 `fulfils(...)` 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:

```
leak(App, MindSet) :-
    violated(data_authenticity(App, Report), 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 NET.SNIFF.LEAK.HTTP.

```
192.168.1.117/24 > 192.168.1.3 >> [14:10:21] [net.sniff.leak.http]
    http local POST example.com Mozilla/5.0 (X11; Android arm; rv:63.0)
        Gecko/20100101 Firefox/63.0

  Method: POST
  URL: /
  Headers:
    Host: example.com
    User-Agent: Mozilla/5.0 (X11; Android arm; rv:63.0)
        Gecko/20100101 Firefox/63.0

  Form:
    mgtxt => This message will be intercepted.
    sendbtn =>   Send
    nmtxt => U. N. Owen
    action => send
```

while Figure 3 offers a visualisation of a subset of the translated report (i.e., a subset of the instantiated ASO).

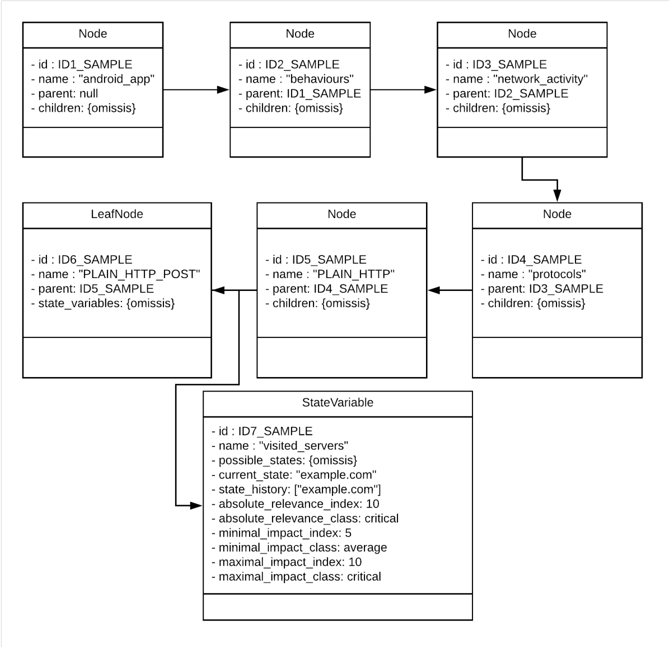
Figure 2 depicts a schematic visualisation of the phases.

Figure 4 shows a visualisation of a subset of the raw report from AndroTotal (in the form of a screenshot of a web page for presentation reasons), while Figure 5 represents a visualisation of a subset of its translated form.

After the aggregation, the relevant subsection of the technological knowledge base we derive from the the final report is the following:

```
%static part of the knowledge base.

root(Report, rootID, android_app, null, ChildrenIDs) :-
    node(Report, rootID, android_app, null, ChildrenIDs).

child(Report, NodeID, ParentID) :-
    node(Report, NodeID, _, ParentID, _).

child(Report, NodeID, ParentID) :-
    leaf(Report, NodeID, _, ParentID, _).

ancestor_of(Report, NodeID, AncestorID) :-
    child(Report, NodeID, AncestorID).

ancestor_of(Report, NodeID, AncestorID) :-
    node(Report, NodeID, _, ParentID, _),
    ancestor_of(Report, ParentID, AncestorID).

violated(data_confidentiality(App, Report), skeptical, rule1) :-
    root(Report, RootID, _, _, _),
    child(Report, MetID, RootID),
    child(Report, BehID, RootID),
    child(Report, NetworkID, BehID),
    child(Report, ProtID, NetID),
    child(Report, HttpID, ProtID),
    child(Report, PostID, HttpID),
    state_var(Report, AppHashID, MetID, app_hash, App, _, _, _, _, _, _, _),
    state_var(Report, VisitedID, PostID, visited_servers,
              _, History, _, _, _, _, Maii, _),
    Maii > 5,
    \+History == [].
```

[3] package name: com.android2.calculator3



Node
- id : ID1_SAMPLE
- name : "android_app"
- parent: null
- children: {omissis}

Node
- id : ID2_SAMPLE
- name : "behaviours"
- parent: ID1_SAMPLE
- children: {omissis}

Node
- id : ID3_SAMPLE
- name : "network_activity"
- parent: ID2_SAMPLE
- children: {omissis}

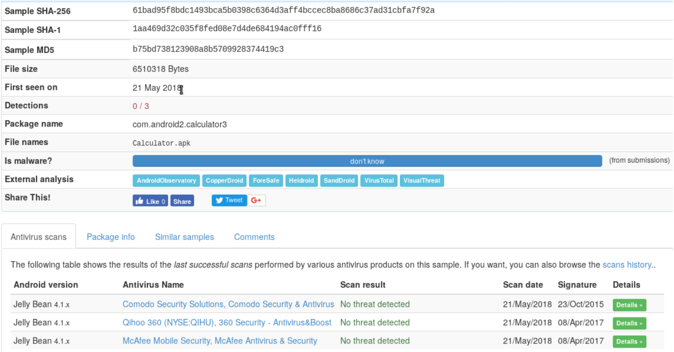
LeafNode
- id : ID6_SAMPLE
- name : "PLAIN_HTTP_POST"
- parent: ID5_SAMPLE
- state_variables: {omissis}

Node
- id : ID5_SAMPLE
- name : "PLAIN_HTTP"
- parent: ID4_SAMPLE
- children: {omissis}

Node
- id : ID4_SAMPLE
- name : "protocols"
- parent: ID3_SAMPLE
- children: {omissis}

StateVariable
- id : ID7_SAMPLE
- name : "visited_servers"
- possible_states: {omissis}
- current_state: "example.com"
- state_history: ["example.com"]
- absolute_relevance_index: 10
- absolute_relevance_class: critical
- minimal_impact_index: 5
- minimal_impact_class: average
- maximal_impact_index: 10
- maximal_impact_class: critical

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 *omissis*.



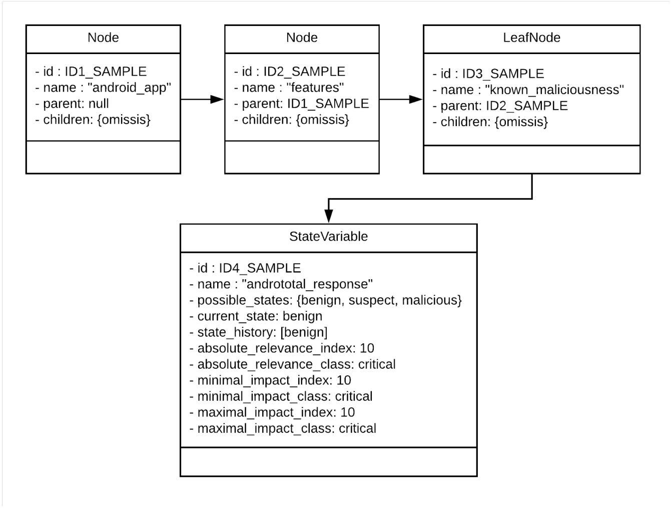
Sample SHA-256	61bad95f8bdc1493bca5b0398c6364d3aff4bccec8ba8686c37ad31cbfa7f92a
Sample SHA-1	1aa469d32c035f8fed08e7d4de684194ac0fff16
Sample MD5	b75bd738123908a8b5709928374419c3
File size	6510318 Bytes
First seen on	21 May 2018
Detections	0 / 3
Package name	com.android2.calculator3
File names	Calculator.apk
Is malware?	don't know (from submissions)
External analysis	AndroidObservatory CopperDroid ForeSafe Heldroid SandDroid VirusTotal VisualThreat
Share This!	Like 0 Share Tweet G+

The following table shows the results of the *last successful scans* performed by various antivirus products on this sample. If you want, you can also browse the scans history..

Android version	Antivirus Name	Scan result	Scan date	Signature	Details
Jelly Bean 4.1.x	Comodo Security Solutions, Comodo Security & Antivirus	No threat detected	21/May/2018	23/Oct/2015	Details »
Jelly Bean 4.1.x	Qihoo 360 (NYSE:QIHU), 360 Security - Antivirus&Boost	No threat detected	21/May/2018	08/Apr/2017	Details »
Jelly Bean 4.1.x	McAfee Mobile Security, McAfee Antivirus & Security	No threat detected	21/May/2018	08/Apr/2017	Details »

Figure 4: A visualisation of the report from AndroTotal

```
% from the final report

node(reportID, rootID, android_app, null, [behID, featID, metID,  ...]).
node(reportID, behID, behaviours, rootID, [netID, ...]).
node(reportID, netID, network_activity, behID, [protID, ...]).
node(reportID, protID, protocols, netID, [httpID, ...]).
node(reportID, httpID, "PLAIN_HTTP", protID, [hpostID, ...]).

leaf(reportID, hpostID, "PLAIN_HTTP_POST", httpID, [visitedID, ...]).

node(reportID, featID, features, rootID, [malicID, ...]).

leaf(reportID, malicID, known_maliciousness, featID, [responseID, ...]).

leaf(reportID, metID, metadata, rootID, [appHashID, ...]).

state_var(reportID, appHashID, metID, app_hash, actual_hash,
          [actual_hash], 10, critical, 10, critical, 10, critical).

state_var(reportID, visitedID, hpostID, visited_servers, "example.com",
          ["example.com"], 10, critical, 5, average, 10, critical).

state_var(reportID, responseID, malicID, andrototal_response, benign,
          [benign], 10, critical, 10, critical, 10, critical).
```



Node
- id : ID1_SAMPLE
- name : "android_app"
- parent: null
- children: {omissis}

Node
- id : ID2_SAMPLE
- name : "features"
- parent: ID1_SAMPLE
- children: {omissis}

LeafNode
- id : ID3_SAMPLE
- name : "known_maliciousness"
- parent: ID2_SAMPLE
- children: {omissis}

StateVariable
- id : ID4_SAMPLE
- name : "andrototal_response"
- possible_states: {benign, suspect, malicious}
- current_state: benign
- state_history: [benign]
- absolute_relevance_index: 10
- absolute_relevance_class: critical
- minimal_impact_index: 10
- minimal_impact_class: critical
- maximal_impact_index: 10
- maximal_impact_class: critical

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 *omissis*.

from which, through reasoning, we are able to derive that our test app violates the property of data confidentiality. Plugging this result into the legal knowledge base, and performing additional reasoning shows that our test application does not fulfil one of the obligations (the very same data confidentiality).

Interpreting the results. Using the technological knowledge base we derived that the property of data confidentiality does not hold for our test application. We have used the technological knowledge base to determine the truth value of one of the concepts introduced, but not explained by the law. Then, we propagated this finding to the legal knowledge base, and we found that, because of that, our test application does not comply with one of the relevant obligations. We have, in other words, found an incompatibility between what the law prescribes, and one of the behaviours of our test application.

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 application 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

[1] [n.d.]. 2018 UK Data Protection Act. https://www.legislation.gov.uk/ukpga/2018/12/contents/enacted Available at https://www.legislation.gov.uk/ukpga/2018/12/contents/enacted.

[2] [n.d.]. Android Market Share. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems Available at https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems.

[3] [n.d.]. The Network and Information Systems Regulations 2018. http://www.legislation.gov.uk/uksi/2018/506/made Available at http://www.legislation.gov.uk/uksi/2018/506/made.

[4] [n.d.]. Security | Android Open Source Project. https://source.android.com/security Available at https://source.android.com/security.

[5] [n.d.]. The Transport Layer Security (TLS) Protocol Version 1.3. https://tools.ietf.org/html/rfc8446 Available at https://tools.ietf.org/html/rfc8446.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[7] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis.* ACM, 27–38.

[8] Trevor Bench-Capon. 1997. Argument in Artificial Intelligence and Law. *Artificial Intelligence and Law* 5, 4 (Dec 1997), 249–261.

[9] Trevor Bench-Capon and Marek Sergot. 1989. Towards a Rule Based Representation of Open Texture in Law. In *Computing Power and Legal Reasoning*, Charles Walter (Ed.). Greenwood Press, Chapter 6, 39–60.

[10] Michael Bierma, Eric Gustafson, Jeremy Erickson, David Fritz, and Yung Ryn Choe. 2014. Andlantis: Large-scale Android dynamic analysis. *arXiv preprint arXiv:1410.7751* (2014).

[11] Stefano Bromuri and Kostas Stathis. 2008. Situating cognitive agents in GOLEM. *Engineering environment-mediated multi-agent systems* (2008), 115–134.

[12] Keith L. Clark. 1977. Negation as Failure. In *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977. (Advances in Data Base Theory)*, Hervé Gallaire and Jack Minker (Eds.). Plemum Press, New York, 293–322.

[13] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. RFC Editor. http://www.rfc-editor.org/rfc/rfc5280.txt http://www.rfc-editor.org/rfc/rfc5280.txt.

[14] Anthony Desnos et al. 2011. Androguard. *URL: https://github. com/androguard/androguard* (2011).

[15] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)* 44, 2 (2012), 6.

[16] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[17] 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union* L119 (4 May 2016), 1–88. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC

[18] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 50–61.

[19] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 576–587.

[20] Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. 2014. Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications. *arXiv preprint arXiv:1402.4826* (2014).

[21] Lance J Hoffman. 2012. *Building in big brother: the cryptographic policy debate*. Springer Science & Business Media.

[22] Hideaki Ishii and Roberto Tempo. 2014. The PageRank problem, multiagent consensus, and web aggregation: A systems and control viewpoint. *IEEE Control Systems* 34, 3 (2014), 34–53.

[23] Antonis C. Kakas, Paolo Mancarella, Fariba Sadri, Kostas Stathis, and Francesca Toni. 2008. Computational Logic Foundations of KGP Agents. *J. Artif. Intell. Res. (JAIR)* 33 (2008), 285–348.

[24] S Karthika, S Gunanandhini, and Mr A Vijayanarayanan. 2013. Android Based Effective and Efficient Search Engine Retrieval System Using Ontology. *IJREAT International Journal of Research in Engineering & Advanced Technology* 1, 1 (2013).

[25] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014 Third International Workshop on*. IEEE, 3–17.

[26] Federico Maggi, Andrea Valdi, and Stefano Zanero. 2013. AndroTotal: a flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*. ACM, 49–54.

[27] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE bites: exploiting the SSL 3.0 fallback. *Security Advisory* (2014).

[28] Juan Antonio Morente-Molinera, Robin Wikström, Enrique Herrera-Viedma, and Christer Carlsson. 2016. A linguistic mobile decision support system based on fuzzy ontology to facilitate knowledge mobilization. *Decision Support Systems* 81 (2016), 66–75.

[29] Nils J. Nilsson. 1994. Teleo-reactive Programs for Agent Control. *J. Artif. Int. Res.* 1, 1 (Jan. 1994), 139–158.

[30] H. Prakken and G. Sartor. 1997. *A Dialectical Model of Assessing Conflicting Arguments in Legal Reasoning*. Springer Netherlands, Dordrecht, 175–211.

[31] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. 2018. BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews. In *Research in Attacks, Intrusions, and Defenses*, Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis (Eds.). Springer International Publishing, Cham, 25–46.

[32] Pedro Sánchez, Bárbara Álvarez, Ramón Martínez, and Andrés Iborra. 2017. Embedding statecharts into Teleo-Reactive programs to model interactions between agents. *Journal of Systems and Software* 131 (2017), 78–97.

[33] Ferial Shayeganfar, Amin Anjomshoaa, and A Min Tjoa. 2008. A smart indoor navigation solution based on building information model and google android. In *International Conference on Computers for Handicapped Persons*. Springer, 1050–1056.

[34] Kerry-Louise Skillen, Liming Chen, Chris D Nugent, Mark P Donnelly, and Ivar Solheim. 2012. A user profile ontology based approach for assisting people with dementia in mobile environments. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*. IEEE, 6390–6393.

[35] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSSâĂŽ14*. Citeseer.

[36] Kostas Stathis. 2000. A Game-based Architecture for Developing Interactive Components in Computational Logic. *Journal of Functional and Logic Programming* 2000, 5 (March 2000).

[37] Kostas Stathis and Marek Sergot. 1996. Games as a Metaphor for Interactive Systems. In *People and Computers XI*, Martina Angela Sasse, R. Jim Cunningham, and Russel L. Winder (Eds.). Springer London, London, 19–33.

[38] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.. In *NDSS*.

[39] Francesca Toni, Mary Grammatikou, Stella Kafetzoglou, Leonidas Lymberopoulos, Symeon Papavassileiou, Dorian Gaertner, Maxime Morge, Stefano Bromuri, Jarred McGinnis, Kostas Stathis, Vasa Curcin, Moustafa Ghanem, and Li Guo. 2008. The ArguGRID Platform: An Overview. In *Grid Economics and Business Models*, Jörn Altmann, Dirk Neumann, and Thomas Fahringer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 217–225.

[40] Edgaras Valincius, Hai H Nguyen, and Jeff Z Pan. 2015. A Power Consumption Benchmark Framework for Ontology Reasoning on Android Devices.. In *ORE*. 80–86.

[41] Johann Vincent, Christine Porquet, Maroua Borsali, and Harold Leboulanger. 2011. Privacy protection for smartphones: an ontology-based firewall. In *IFIP International Workshop on Information Security Theory and Practices*. Springer, 371–380.

[42] Douglas Walton. 2005. *Argumentation methods for artificial intelligence in law*. Springer Science & Business Media.

[43] Mark Witkowski and Kostas Stathis. 2004. A Dialectic Architecture for Computational Autonomy. In *Agents and Computational Autonomy*, Matthias Nickles, Michael Rovatsos, and Gerhard Weiss (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 261–273.

[44] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *NDSS*, Vol. 16. 21–24.

[45] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.. In *USENIX security symposium*. 569–584.

[46] Roberto Yus, Carlos Bobed, Guillermo Esteban, Fernando Bobillo, and Eduardo Mena. 2013. Android goes Semantic: DL Reasoners on Smartphones.. In *Ore*. Citeseer, 46–52.

[47] Min Zheng, Mingshen Sun, and John CS Lui. 2014. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*. IEEE, 128–133.