Automated Analysis of General Data Protection Regulation Violations in WordPress Plugins

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Patrick Thomas
Spring, 2021

Technical Project Team Members
Yinzhi Cao
Michelangelo van Dam
Mingqing Kang
Faysal Hossain Shezan
Zihao Jerry Su
Yuan Tian
Erwin Wijaya

Signature ____ Date __________
Patrick Thomas

Approved ____ Date __________
Yuan Tian, Department of Computer Science

Abstract

Recent regulations like the General Data Protection Regulation (GDPR) have introduced requirements as to how software applications have to handle personally identifiable information. Methods like static code analysis have shown promise in identifying security and privacy violations. This project aims to apply static code analysis to WordPress plugin analysis, which requires cross-language analysis for the programming languages present in WordPress plugins: PHP, HTML, JavaScript, and SQL. In addition, the static analysis attempts to identify GDPR violations that can be identified from plugin source code alone: the right to data access, the right to data deletion, and features like cookie consent. Early prototypes of the analysis tool showed promise in finding data flows from source to sink but struggled with false positives when ran on 2,060 WordPress plugins. Later iterations of the tool are more powerful and can abstract and map plugin source code to GDPR requirements, including specifying what lines of code necessitate which GDPR requirements. The project is nearly complete and still requires final tweaks and evaluation on all 35,274 plugins available on WordPress.org as of August of 2020.

Introduction

The General Data Protection Regulation (GDPR) is a recent European Union law framework that aims to support data rights as well as support digital privacy, as motivated by an increase in digital privacy awareness and events like data breaches. While providing clear benefits to individuals such as the right to data access, right to data deletion, proper data security, and data collection consent, there are challenges in designing GDPR-compliant applications and websites. Knowledge as to whether a website is truly in compliance can be difficult to gather, as there are myriad ways in which, for example, user data is saved in an unstructured way and is hard for administrators to find and remove upon request. A multitude of free and paid services exist to evaluate the GDPR compliance and vary in complexity, ranging from consulting to cookie analyzers to do-it-yourself checklists. These all either only cover a very specific subset of GDPR requirements like cookie consent or involve slow or expensive manual review. Similarly, these solutions vary in the amount of information that they need to determine GDPR compliance, where checking for cookies or a proper privacy policy requires only public-facing information, but data security would involve a comprehensive review of both possibly private source code and infrastructure. One such solution and the goal of this project is to develop an automated code analysis tool that takes web application code as input and outputs a list of GDPR requirements, which avoids costly manual analysis.

Another challenge in determining application GDPR compliance is the variety of languages involved in a typical web application, like HTML, PHP, and JavaScript. Tools like PHP Joern and php2ast can convert a PHP project into an abstract syntax tree and control flow graph, capable of statically capturing the control flow of a PHP web application (Backes et al., 2017; Popov, 2014/2021). However, these technologies are limited to PHP, and any personal information transmitted to other languages including client-side languages like JavaScript can then be transmitted elsewhere, completely avoiding detection through PHP code analysis. Furthermore, a combination of JavaScript and HTML may display and record a cookie consent banner entirely outside of a PHP script, rendering this type of requirement unable to be checked with static PHP analysis alone. Both PHP and JavaScript can edit the HTML document-object model, complicating the analysis of HTML concerning what information is displayed to a user. Thus, to fully model a web application, a cross-language approach is necessary to capture all possible explicit logic.

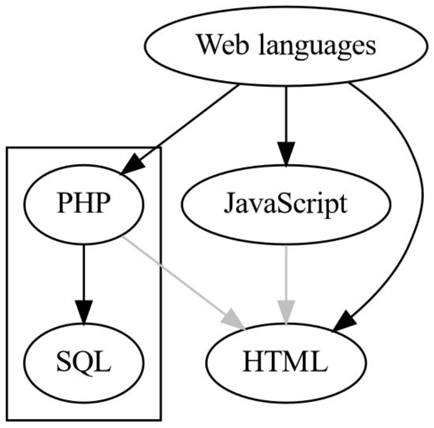

Figure 1: Model of how web application programming languages interact with each other. Both PHP and JavaScript can inform the HTML document object model, as represented by the gray edges. PHP code also can contain SQL queries, requiring an additional language in the web application model.

Background

At the core of static analysis are abstract syntax trees (ASTs) and code property graphs (CPGs). An AST augmented with a CPG forms an easy-to-search representation of multiple programming languages in a single graph database. CPGs also include control flow graphs (CFGs), which encode control flow and branching through the nodes of an AST. Combined with CFGs, this structure provides a way to statically follow the control flow of a program with easy access to code details. In addition to regular program flow, CFGs also include function and method call edges, where a function call creates a directed edge between the function or method call to the declaration.

Aside from control flows, CPGs also can include the flow of data and information from node to node. An example of this assigning the value of a variable to another variable, which would be represented in our combined AST/CPG as an edge between the two variables. Thus, an AST and CPG together represent the source code, branches, program flow, and simple data dependencies.

Related Work

Static code analysis is a widely studied topic, and it likewise has been applied to a variety of situations, including code correctness, security, and privacy. JFlow was an early application of static analysis to security and privacy for Java code (Myers, 1999). One particularly popular area for privacy static analysis is Android apps. AndroRisk is a risk quantifier for Android apps based on their requested device permissions along with investigation of an app's used libraries (Sarma et al., 2012). Kim et al. (2012) raise concerns that permission-based approach to Android app evaluation is not sufficient; in response they developed ScanDal, which statically analyzes

Android apps for privacy leaks via tracking data flows from sinks to sources (Kim et al., 2012). Xiao et. al (2015) applied static analysis to TouchDevelop scripts for Android, and these scripts were analyzed for private information sources flowing to sinks to outside sources as well as the security and privacy of such information flows (Xiao et al., 2015). Their method was able to identify that 14.29% of 546 total scripts had private information sources flowing to a sink and that only 10.1% were truly unsecured, as in the other 4.19% of apps were considered safe.

Ferrara and Spoto (2018) discussed the direct application of static analysis to GDPR compliance, approaching the problem formally and through the multiple technical and business groups involved in web application development (developers, project managers, data protection officers, etc.) (Ferrara & Spoto, 2018). Ferrara et. al (2018) also discussed the application of taint analysis, an extension of static analysis, as it related to GDPR (Ferrara et al., 2018).

One notable limitation of the earlier approaches to static or mostly static code analysis was that it only captures information flows in a single programming language. BridgeTaint is one system that aims to perform a cross-language taint analysis and hybrid Web and Android native apps (Bai et al., 2019). Their approach included the identification of private information leaks, and they experimentally identified that 85.7% of the 1055 popular Android apps had some sort of security or privacy risk. Mandal et. al (2020) applied cross-program taint analysis to IoT systems to find information leaks from IoT devices and systems (Mandal et al., 2020).

System Design

The system was designed to take a WordPress plugin as input and output a list of relevant GDPR areas, specific lines or data flows that affected that item's inclusion, and finally whether those areas are GDPR-compliant or not. These GDPR areas were the right to deletion, right to access, right to consent, as well as security function usage. We chose these categories since other GDPR compliance goals are not evaluable with WordPress plugin source code; for example, it is impossible to tell whether or not the developer properly masked their test data, where the failure to do so constitutes a GDPR violation, from the plugin source code alone.

System Architecture

To accomplish the goal of analyzing WordPress plugins, a variety of tools were employed or developed. These tools processed a WordPress plugin into ASTs, represented the AST in an accessible format, and performed analysis and manipulations on the AST to produce actionable and specific GDPR compliance results. On top of this smaller architecture was the larger batch analysis architecture that oversaw the running the tool on thousands of WordPress plugins.

We chose to develop the majority of the project in Python since it allows for easier development as compared to lower-level languages and is widely supported. This wide support is useful for connecting with the graph database, for example. While Python may not necessarily be as fast as lower-level languages, this does not greatly affect the project because of latency with communicating with the graph database and large queries on the database taking most of the execution time of the program. Aside from Python, Bash scripts are utilized for automation, and some JavaScript programs were developed to interface with libraries to generate JavaScript ASTs.

On a broad level, analyzing an individual plugin involved first converting a plugin into ASTs and CPGs, performing some processing on the graph to determine sinks or sources as well as improve the overall CPG's usefulness, and finally perform some analysis on the results from the aforementioned processes. This is reflected in the figure below.

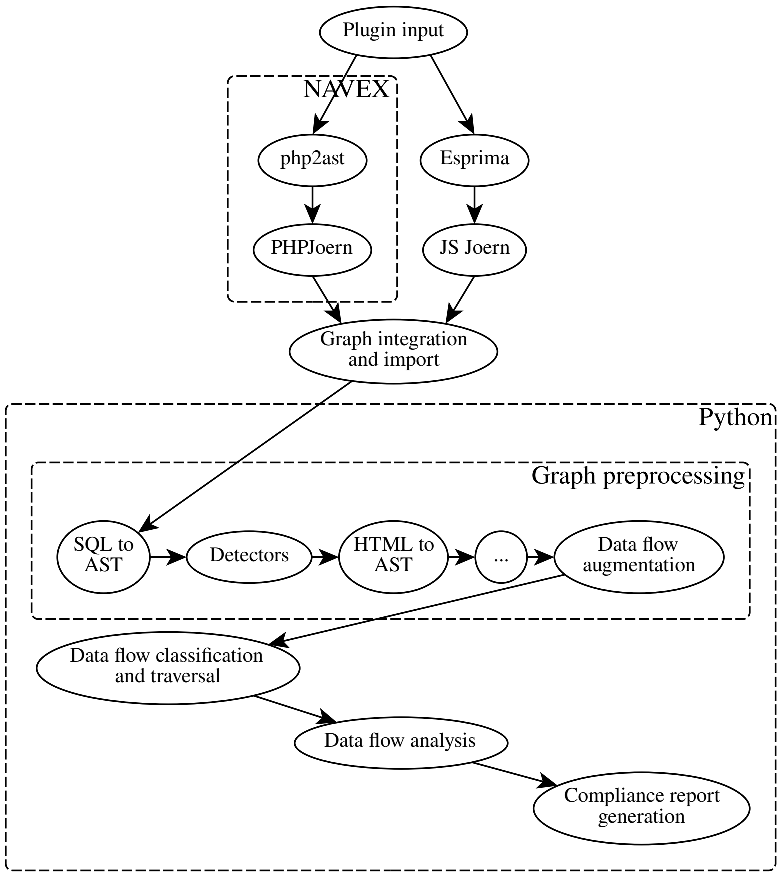

Figure 2: Overall view of the major stages of analyzing a single plugin. The edges represent dependencies and do not necessarily represent the order in which these stages were executed. Nodes within the "Python" block are ran inside of the developed main analysis tool. Other nodes mainly involve generating and combining ASTs.

Since WordPress and WordPress plugins are backed by PHP, we needed to convert the PHP source code within a plugin into an AST and CPG. This was done via NAVEX (Alhuzali et al., 2018). While NAVEX is a exploit and vulnerability generation tool for web applications, it analyzes PHP applications via ASTs and CPGs, meaning that the input stage from NAVEX is suitable for use towards our goal of static analysis. NAVEX itself relies on PHPJoern and php2ast, which is primarily what our analysis tool borrows from NAVEX (Popov, 2014/2021; Skoruppa, 2017/2021). Similarly, to capture front-end logic in JavaScript, all JavaScript source code files were converted into an AST via Esprima, and then a JavaScript extension for Joern was used to produce a CPG for the JavaScript AST (Hidayat, 2011/2021; *Joernio/Joern*, 2019/2021). Finally, the PHP and JavaScript ASTs and CPGs were combined and then imported into a graph database utilizing Neo4j (Neo4j, Inc., 2021).

While PHP and JavaScript ASTs are generated outside of the main analysis tool, HTML and SQL ASTs are generated during the analysis tool's execution, shown in the "Graph preprocessing" stage within the "Python" block in the figure above. HTML source code is extracted from the PHP AST (in the form of implicit or explicit echo statements). To do this, the full PHP AST and CPG need to be generated and loaded into the graph database to find echo statements, determine the order in which they are executed, and statically evaluate their content.

In addition to the plugin-level architecture, another challenge in evaluating the analysis tool was scale, as there are thousands of WordPress plugins available online as of August of 2020. Since the analysis tool relies on multiple tools outside of the code that we developed, the process to evaluate multiple plugins is not straightforward. This architecture is represented in the figure below.

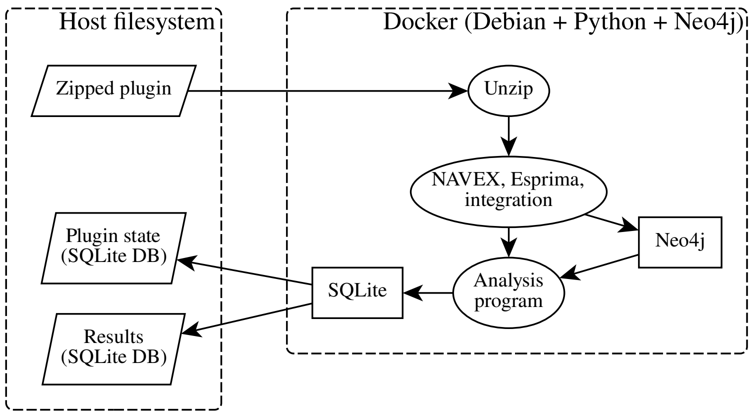

Figure 3: Example of the large-scale, batch analysis architecture on thousands of WordPress plugins. Note that an arbitrary number of identical Docker containers are launched at the same time while the "Host filesystem" resources are shared between all Docker containers.

To efficiently utilize machine resources as well as make the application portable, the individual plugin architecture is containerized via Docker (Docker, Inc., 2021). Each Docker container takes a plugin from a shared directory on a filesystem, converts the plugin into ASTs/CPGs, loads the AST into the graph database, analyzes it, and stores the result in a SQLite database on the shared filesystem. Once this is done, the container then obtains another not analyzed plugin and repeats this process. Since the container both runs tools (which are possibly multi-threaded or executed in parallel) and a database server in the background, the container best utilizes resources on a multi-core system.

The plugin organization and results databases are shared between containers and stored on the shared filesystem. Thus the containers take plugins as they are completed and automatically divide the plugins among themselves as the plugins are executed. The plugin organization database keeps track of the basic status of the execution of the analysis tool on a plugin, ensuring that plugins are only fully analyzed once when multiple analysis tools are running. If any part of the analysis tool fails on a plugin, the analysis tool will either attempt to rerun up to three total times (in the event of graph database or analysis tool problems) or skip the plugin (on NAVEX, Esprima, Neo4j loading, or AST integration errors).

Technical Implementation

Many technical challenges arose during the design and construction of the analysis tool, and these typically involved creating more useful substructures within the AST or developing more efficient ways in which to communicate with the database server. Given the scope of the project, this section's focus will be primarily on my contributions to the project. Some other notable challenges, interesting problems, or core algorithms and components are still included, however.

Database Loading, Integrity, and Optimizations Given the number of sources for and potentially large size of the Neo4j graph database, special care had to be taken to ensure the integrity of the database. Neo4j includes a batch import tool that can import large numbers of nodes and edges from CSV files, but internal node IDs are required to be unique. Furthermore, we get multiple CSV files from NAVEX and Esprima with overlapping node ID spaces (both are indexed starting at zero). Those nodes are referenced according to their IDs in their corresponding edge and CPG edge CSVs, which leads to collisions in node IDs. Edges reference their own node ID spaces, meaning that the node IDs in edges also have to be reassigned. A Python script was developed to merge nodes CSVs. This automatically reassigns IDs of each node and all edges that reference a node. A very simple diagram of how this script operates is also shown below.

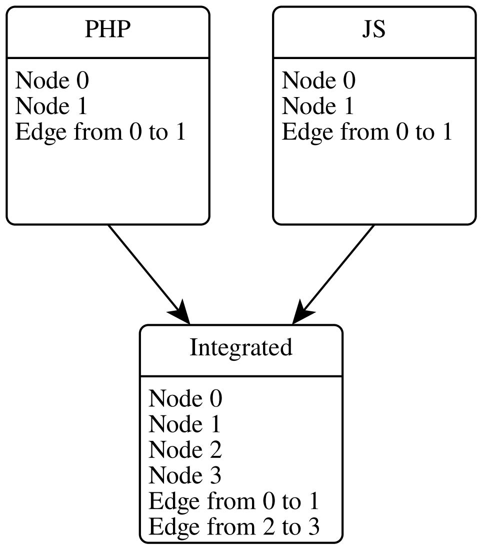

Figure 4: Simple example of the node collision resolution script. In practice, each node and edge include much more information and labels as to what the nodes are such that all data are still identifiable and usable.

Several properties of nodes were also indexed to generally improve the database's performance on random node lookups, which are common in the analysis tool. Neo4j provides interfaces to do this, and indices were created on node IDs and text value.

In some rare cases, Esprima and JS Joern generate AST or CPG edges that reference nodes that do not exist. The integration script takes this into account and removes these edges. Leaving the edges as-is was not viable, as Neo4j's batch import tool exits if there are too many erroneous edges. Alternatively the nodes could have been recreated to preserve the tree structure of the AST/CPG, but any information in these nodes would still have been lost.

WordPress Modeling Given that we are analyzing WordPress plugins, we need to model a significant number of WordPress interfaces to correctly know what data a WordPress function changes and returns. While WordPress ultimately communicates with a relational database which could be discovered by static analysis, we chose not to include the entire WordPress environment when analyzing a plugin. This decision was made given the size of WordPress; analyzing different plugins would require analyzing the WordPress source code each time, significantly slowing the analysis of any plugin. We rather modeled any interfaces that WordPress provides that are interesting to the analysis, as this avoids the costly operation of analyzing WordPress multiple times.

To model WordPress functions and methods, we scraped WordPress.org's documentation on their functions. This information includes parameter types and return types. We augmented this information with our manual labeling of those functions as to whether they store or retrieve personal information and what particular information is being stored (user, post metadata, etc.). Some other set-based attributes were added to the function and method information, allowing us to do set arithmetic to determine if any information was left not deleted, for example.

Other team members did a significant amount of work modeling WordPress's action hook system. This system dynamically calls functions by name upon a trigger or manual activation, and those action hooks return some data when called manually. Since this functionality is dynamic (actions are called by its registered name and thus is not static by nature), we modeled it as there are well-documented interfaces to add these hooks and triggers. This data is sometimes personal information and thus had to be modeled to obtain a comprehensive understanding of how a plugin handles personal information.

Detector Design and Architecture Another challenge in designing the analysis tool was developing a scalable method in which to identify the usage of interesting functions within an AST. Sinks, sources, and privacy-enhancing or security functions need to first be identified to perform the static analysis. Functions of importance were partially identified in the manual labeling described earlier, but there is still the problem of those functions within the AST. This problem was approached by grouping similar functions and methods and writing a *detector* for each group. Each detector looks its functions usages, gathers some knowledge about how that function is being used like passed parameters, and records each found function to its *findings* set. Detectors were modularized in this way since function groups often need special care in analyzing their usage. Database operations often have an associated SQL query, and a PHP cURL session involves multiple function calls that specify different details about that session like whether the session is encrypted or not. The *detector manager* runs those detectors via a thread pool and also automatically discovers all written detectors upon their import in the code. The detector manager also aggregates findings and provides interfaces to quickly lookup those results, like whether or not a specific node falls under any findings or not. Thus, the detector, finding, and detector manager system provides a scalable and easily extensible way to analyze individual function or method usages and automatically incorporate those into later analyses.

PHP to HTML As mentioned earlier, one part of the program determines the HTML output of the PHP source code via echo statements. We want to eventually have an HTML AST such that we can determine what information is being displayed to the user or other users as well as what information an HTML form may collect. In addition, knowing the HTML displayed to the user is useful in determining what, if any, cookie consent or similar messages are shown to the user.

The first problem within this is encountering all PHP echo statements in order. With the entry point of each PHP file, we traverse the CFG to obtain all function or method calls and echo statements in order of execution. For every function or method traversed, we also perform the same algorithm on the CFG of the function's definition. Once all functions and methods are traversed, we then have a representation of the exact order than nodes and thus lines of code are executed. This discovery operation was done recursively as opposed to gathering all possible CFG and call information with a single database operation. The latter has performance issues in Neo4j on complex control flows involved lots of branching and calls and necessitated a better algorithm.

Another challenge is in determining the text content of individual echo statements when building the HTML AST. Python's own HTML parser was used to convert HTML code into an AST, which allowed for a fast, native, and dependency-free way to create HTML ASTs during runtime. To determine echo statement output, we developed functions to recursively browse a node's children and return the text content of the node where possible. Certain operations, like binary string concatenations, were evaluated and completed with the statically known arguments to the operation. Variables were either statically resolved if possible or substituted with their name. Once the recursive function returns, we essentially have a string containing the echo statement's output or theorized output. This string is then fed to the HTML parser. Since the HTML parser is fed information per echo statement and outputs node information in response to that input, we can match how variables influence the resulting HTML AST as nodes are created with relative ease. This is represented with the edge between the PHP AST and HTML AST in the figure below. The entire process described here is also displayed in the figure below.

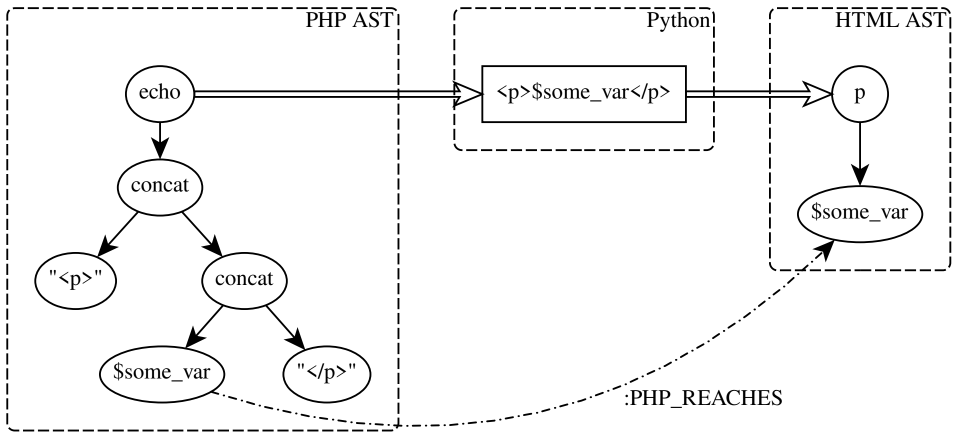

Figure 5: Abstracted example of how a PHP echo statement is converted into an HTML AST. Note the edge between $some_var in the PHP and HTML groups; this edge is created in the graph database and allows us to precisely know what PHP variables dynamically influence the HTML content. Constant strings, like "<p>", were not connected in the HTML AST since their non-dynamic behavior was assumed to be irrelevant to personal information flows.

PHP to SQL A similar problem to the PHP to HTML problem was determining SQL operations sent to the WordPress database or other databases. SQL operations, much like echo statements represented in the figure above, often involved binary concatenation for plugin-specific table names. Once the SQL query was statically determined via the same previously discussed recursive algorithm, it was converted into a SQL AST via sqlparse, a Python package for parsing SQL queries (Albrecht, 2012/2021). The resulting AST was added to the graph database, and additional edges were added that represented the sequential flow of statements in the SQL query, similar to flow in a CFG. These SQL ASTs are crucial in determining what tables a plugin might create (including the fields of said tables), what tables or fields a query adds, updates, or removes from, and what tables are deleted when a plugin is uninstalled or disabled.

Data Flow Analysis The final stage of the analysis tool analyzes the found data flow paths that are suspected to contain personal information. Each path is evaluated as to whether it violates GDPR in the following areas: access, consent, and deletion. These software requirements were mapped to a series of functions that determine the need for plugin compliance and actual compliance to that requirement, as shown in the figure below. This architecture allows for relevant GDPR areas to be identified; for example, if a plugin has no personal information sinks, then deletion is not required as there is never any personal information saved to the server. The modularization of these requirements also allows for easier backtracking to determine what specifically necessitated a requirement. The requirement functions made extensive usage of the detector interfaces explained earlier.

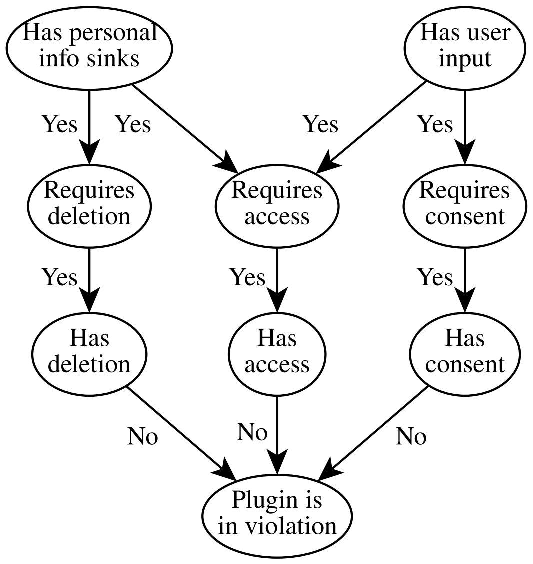

Figure 6: Graph showing dependencies on determining whether a plugin constitutes a GDPR violation or not. "Plugin is in violation" is true if there is any path from the top nodes to that node.

Caching The analysis tool very frequently communicates with the graph database. Even though the graph database is always hosted on the same local machine, those queries still add a significant amount of overhead in communicating the query to the database, processing and parsing the query, determining the result, sending the result back to the Python program, and unmarshalling the response. Even though this might typically take on the order of several milliseconds on a sufficiently powerful machine for a very simple query, a recursive function to resolve the value of some echo statement might make hundreds of queries to understand the output of the echo statement. Given the frequency of echo statements and other similar situations, the analysis tool can begin to take several hours to analyze a plugin. Thus, node, edge, and node property caching are required for the analysis tool to function correctly.

The analysis tool's caching is based on spatial locality: nodes and their neighbors were cached under the assumption that the node's neighbors would be accessed. For example, one commonly used function is one to look up a single node via that node's ID. This result can be cached, especially after the graph has been fully augmented. Since neighbors to the node are often accessed next, the query to retrieve this node also returns all neighbors within two degrees of the node, and all of the nodes are cached according to their ID. Now, subsequent queries on that node or its close neighbors return nearly instantaneously because of the caching. This also makes operations like getting all of a node's children's IDs much faster since, again, the database result is already known and thus doesn't need to be queried.

This caching relies on the assumption that our graph is not changing or at least does not change often. This assumption generally holds outside of augmenting the graph; early in the tool's execution we add the HTML and SQL ASTs, but afterward, nodes and their properties generally remain constant. In cases where we modify a node that is likely already cached, we simply have to purge the cache.

Results

The analysis tool is to be evaluated on all 35,274 WordPress.org plugins. This only includes plugins that were hosted online as of August 2nd, 2020. Smaller batch analyses were done on a subset of 2,060 plugins out of the total 35,274 plugins.

The current analysis tool's performance has yet to be evaluated on the entire data set of plugins at the time of writing. Early prototypes from about halfway in the development of the analysis tool were able to identify some data flows within a plugin when analyzed on the smaller set of 2,060 plugins. Results from that analysis performed simple data flow analysis and also aggregated the results of those previously discussed detector objects. Those data flows were evaluated according to whether or not those data flows flowed to a sink and if there were any security functions on the path from source to sink. This early prototype found that 157 of those 2,060 plugins had some sort of personal information flow; however, this prototype was extremely prone to false positives with personal data. On the contrary, for example, some of these plugins were found to connect to analytics URLs, showing that the tools still had some utility in the early stages. The results from the prototype generally did not find personal data flows from sink to source in violation of GDPR, but these data flows often correlated with plugins that used personal information and were in violation of GDPR for other reasons undetected by the tool.

Since then, some components of the analysis tool received major reworks, like the algorithm for finding, classifying, and traversing found data flows. As mentioned earlier, we do not yet have results from a large-scale evaluation on the new tool, either on the smaller subset of 2,060 plugins or the entire plugin set. Anecdotally, the tool is a lot better at finding personal data flows due to better WordPress modeling and the incorporation of the different languages, like JavaScript and HTML. The tool in its current state is much better at providing specific reasoning as to why a plugin is or is not GDPR compliant. Further evaluation still needs to be done to determine the effectiveness of the system.

Conclusions

A significant amount of work towards achieving the research goal has been completed, and only the final steps remain. Our tool automatically performs a cross-language analysis of WordPress plugins concerning GDPR requirements. The analysis tool can successfully combine ASTs and CFGs generated from the multiple programming languages present in a WordPress application: HTML, PHP, SQL, and JavaScript. Early prototypes showed some promise in automatically identifying GDPR violations in WordPress plugins across batch runs. Soon, the final batch evaluation on the completed tool will be done. This work helps privacy-enhancing law like the GDPR be applied to web applications, furthering privacy online.

Future Work

There are several areas in the tool that now, looking back, could have been improved or designed better. While the detector interface accomplishes its goal, it suffers from issues related to retrieving information once it has been gathered due to how generic the interface is, meaning lots of information is stored in a somewhat unstructured manner. This could be improved with a better design that accounts for parameters of interest, parameter types, return types, and a more standard finding or output format.

Another area that would involve a major rework of the tool is a migration from using an external graph database and rather using Python libraries for graph manipulation and analysis. While Neo4j provides powerful ways to query and visualize the data, reliance on an external database server introduces a major performance bottleneck on lots of the analysis and augmentation algorithms that we developed. In addition, Neo4j has poor compatibility with read-only containerization needed for running custom programs on the University of Virginia's High-Performance Computing system, Rivanna, making the program unwieldy to evaluate. While some plugins can span more than 100,000 nodes, the data still does not occupy more than 20MB on a hard drive, so the data is more than capable of being loaded in memory and without a graph database system. The ability of the integration script to fully load all node and edge data into memory further supports that this is feasible with our technology.

References

Albrecht, A. (2021). *Andialbrecht/sqlparse* [Python]. https://github.com/andialbrecht/sqlparse (Original work published 2012)

Alhuzali, A., Gjomemo, R., Eshete, B., & Venkatakrishnan, V. N. (2018). *NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications*. 17.

Backes, M., Rieck, K., Skoruppa, M., Stock, B., & Yamaguchi, F. (2017). Efficient and Flexible Discovery of PHP Application Vulnerabilities. *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 334–349. https://doi.org/10.1109/EuroSP.2017.14

Bai, J., Wang, W., Qin, Y., Zhang, S., Wang, J., & Pan, Y. (2019). BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications. *IEEE Transactions on Information Forensics and Security*, *14*(3), 677–692. https://doi.org/10.1109/TIFS.2018.2855650

Docker, Inc. (2021). *Docker*. https://www.docker.com/

Ferrara, P., Olivieri, L., & Spoto, F. (2018). Tailoring Taint Analysis to GDPR. In M. Medina, A. Mitrakas, K. Rannenberg, E. Schweighofer, & N. Tsouroulas (Eds.), *Privacy Technologies and Policy* (pp. 63–76). Springer International Publishing. https://doi.org/10.1007/978-3-030-02547-2_4

Ferrara, P., & Spoto, F. (2018). *Static Analysis for GDPR Compliance*.

Hidayat, A. (2021). *Jquery/esprima* [TypeScript]. jQuery. https://github.com/jquery/esprima (Original work published 2011)

Joernio/joern. (2021). [Scala]. joern.io. https://github.com/joernio/joern (Original work published 2019)

Kim, J., Yoon, Y., Yi, K., Shin, J., & Center, S. (2012). ScanDal: Static analyzer for detecting privacy leaks in android applications. *MoST*, *12*(110), 1.

Mandal, A., Ferrara, P., Khlyebnikov, Y., Cortesi, A., & Spoto, F. (2020). Cross-program taint analysis for IoT systems. *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 1944–1952. https://doi.org/10.1145/3341105.3373924

Myers, A. C. (1999). JFlow: Practical mostly-static information flow control. *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 228–241. https://doi.org/10.1145/292540.292561

Neo4j, Inc. (2021). *Neo4j*. Neo4j Graph Database Platform. https://neo4j.com/

Popov, N. (2021). *Nikic/php-ast* [PHP]. https://github.com/nikic/php-ast (Original work published 2014)

Sarma, B. P., Li, N., Gates, C., Potharaju, R., Nita-Rotaru, C., & Molloy, I. (2012). Android permissions: A perspective combining risks and benefits. *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, 13–22. https://doi.org/10.1145/2295136.2295141

Skoruppa, M. (2021). *Malteskoruppa/phpjoern* [PHP]. https://github.com/malteskoruppa/phpjoern (Original work published 2017)

Xiao, X., Tillmann, N., Fahndrich, M., de Halleux, J., Moskal, M., & Xie, T. (2015). User-aware privacy control via extended static-information-flow analysis. *Automated Software Engineering*, *22*(3), 333–366. https://doi.org/10.1007/s10515-014-0166-y