This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

Detecting Vulnerabilities in Encrypted Software Code while Ensuring Code Privacy

Jorge Martins, David Dantas, Rafael Ramires, Bernardo Ferreira, *Member, IEEE,*
Ibéria Medeiros, *Member, IEEE*

Abstract—Software vulnerabilities continue to be the primary cause of cyberattacks, largely due to the widespread adoption of web applications, developed for access and management services, that are made available without proper testing. It is crucial to identify vulnerabilities in applications' source code before attackers gain access to them and exploit any vulnerability they may contain. Developers have used static analysis tools to find vulnerabilities in unprotected application code, and software testing companies have started offering software code analysis as a service to assist developers in these findings. Such services require access to unprotected code, which raises concerns about its privacy and intellectual property theft. Moreover, attackers can also perform this analysis using similar tools, if they gain access to the code. It is, therefore, beneficial to have a system that can maintain code privacy by protecting it with cryptographic techniques, while still allowing authorised people to detect vulnerabilities in the encrypted code. This paper presents such a solution, a novel approach to *Software Quality and Privacy* that allows source code to be analysed in a protected manner, preserving its privacy. The proposed solution combines Static Analysis with Searchable Symmetric Encryption (SSE) for confidential vulnerability detection, enabling data and dependency tracking for data flow analysis over encrypted source code. The solution represents the code's data and control flows as an *Encrypted Inverted Index*, in a connected way that enables SSE's queries for vulnerability discovery. The solution was implemented as the CoCoA tool and evaluated experimentally with synthetic and real PHP web applications. Results show that CoCoA has similar precision as standard (non-confidential) static analysis tools – 93% – with real applications, requiring only 209 ms to process 4k lines of code - a modest overhead of 42.7% compared to a non-confidential baseline. With this novel approach, this paper also defines a new research field – *Confidential Code Analysis* –, from which other types of code analysis tasks and approaches can be derived. We believe this work paves the way for this promising area of research, from which *Software Quality and Privacy* can greatly benefit.

Index Terms—Vulnerability Detection, Static Analysis, Code Privacy, Searchable Symmetric Encryption, Software Security Analysis

1 Introduction

Software has played an essential role in our global technological world. Despite the efforts software companies have made to build quality software, the number of vulnerabilities continues to increase [62], so much so that the number of cyber attacks has risen to the highest record in recent years [9]. This increase is justified by the accelerating digital transformation, and indicates that inadequately tested software has been made available to the public, thus causing a decline in overall quality [17]. Web applications are at the root of this transformation, serving as the primary means of accessing and managing services; hence, the vulnerabilities they contain are a source for this security problem. This makes these attractive targets for attackers [66], with SQL injection (SQLi) and cross-site scripting (XSS) being among the most exploited vulnerabilities [64].

To contradict these figures, static analysis tools [36], [45], [48] have been used to analyse unprotected source code, identifying vulnerabilities and suggesting how to fix them. Developers and software testing companies have resorted to these tools to achieve software quality assurance [39], [40], [49], with the latter also starting to offer them as a cloud-based service [58], [61], [67]. In this service, software can undergo a series of code analysis tasks, including vulnerability detection, to properly test the software and improve its quality and reliability [10], [22], [32], [33], [43]. However, this requires access to the unprotected source code, raising concerns about its privacy and theft of intellectual property [50]. Moreover, attackers can also perform this analysis using similar tools if they gain access to the code.

Code obfuscation [4], [31], [37], [60], [70] and code encryption [6], [11], [19] techniques have the goal of making the code unreadable. They are thereby seen as a way for software companies to protect their code against potential copyright violations, reverse engineering, and code privacy preservation, thus preventing (malicious) third parties from inspecting it. When applied, they can transform (i.e., obfuscate/encrypt) the entire code or specific portions (e.g., entry points, sensitive sinks), making it totally or partially illegible for reading. As such, the level of protection offered by these techniques relies on the strength of the algorithms and cryptographic keys used. We have observed that such algorithms have been broken [6], [31] and cryptographic keys have been discovered [11], compromising code protection. Moreover, when the code needs to be analysed (even by developers), it is unprotected (de-obfuscated or decrypted) during the analysis process, which can compromise its secrecy. Hence, although such techniques provide a sense of protection, they do not offer real guarantees of code privacy, and are impractical for third parties to conduct code analysis tasks. An interesting solution to overcome these concerns would be, therefore, to have a system that guarantees the confidentiality of the code by protecting it with cryptographic techniques, while at the same time allowing authorised people to perform code analysis tasks on the encrypted code, such as detecting vulnerabilities.

- *J. Martins, D. Dantas, R. Ramires, B. Ferreira and I. Medeiros are with the LASIGE, DI, Faculdade de Ciências, Universidade de Lisboa - Portugal (e-mail: fc51033@alunos.fc.ul.pt, fc56331@alunos.fc.ul.pt, rframires@ciencias.ulisboa.pt, blferreira@ciencias.ulisboa.pt, and ivmedeiros@ciencias.ulisboa.pt).*

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

This paper presents such a solution, a novel approach that protects software code by using an encryption scheme that enables vulnerability finding on the encrypted code without revealing its contents during the analysis. The approach combines *Static Code Analysis* (SCA) [48] with *Searchable Symmetric Encryption* (SSE) [65], two disjoint research fields that would seem unlikely to be combined. In SCA, the application's code is represented by code property graphs (e.g., control flow graph and data flow graph) [3], [24], [72] which can then be used for different purposes, such as vulnerability detection [16]. Static analysis tools navigate them, looking for what they were programmed for. They may employ various SCA techniques, with taint analysis [36] being the most commonly applied, which allows tracking entry points (user inputs like `$_POST` in PHP) and checking if they reach sensitive sinks (e.g., `echo` for XSS). However, these tools require access to the application's unprotected code, without it being obfuscated nor encrypted. In turn, in SSE [15], text documents are represented through an inverted index [42], where keywords (i.e., the documents' most relevant words) are mapped to the documents where they appear. This index is encrypted in a manner that it leaks no information when stored in remote (untrusted) servers, but nonetheless, authorised clients can query it to find index positions (i.e., the documents) containing the keywords they search for.

We propose combining these two research fields to enable *static analysis capable of operating on and understanding encrypted code*, and implement the approach as a new tool named CoCoA (C*onfidential dete*C*t*i*on of vulner*A*bilities*). CoCoA processes the source code of the developer's application, extracting its data and control flows and building the *Data and Control Flow Graph* (DCFG – a type of program dependence graph [24]). Then it represents the DCFG as an inverted index and encrypts it similarly to SSE, resulting in an *Encrypted Inverted Index* (EII). Subsequently, the EII (i.e., the encrypted DCFG) is used by authorised people to perform static code analysis tasks and find vulnerabilities with confidentiality, by querying the index for specific vulnerability classes. In turn, the queries trigger data flow analysis considering the control flows present in the DCFG.

CoCoA can be used in three distinct scenarios, all with the same purpose – *Software Quality and Privacy* – i.e., when being used, it ensures code privacy and the developer's intellectual property by encrypting the code, while simultaneously allowing its encrypted analysis, which improves the software's quality by identifying vulnerabilities. In the first scenario, developers can use CoCoA only to protect their code and prevent unintended access to it, i.e., without static analysis features. On the second, they can use CoCoA both to protect and test the code against vulnerabilities. Lastly, software testing companies can use CoCoA to offer *Confidential Code Analysis as a Service*. Such a service will, on one hand, allow developers to outsource security analysis tasks to specialised third parties without sacrificing their intellectual property nor the privacy of their code. On the other hand, it will allow testing companies to execute such analysis tasks *out-of-house* rather than the more common yet resource-intensive *in-house* manner (i.e., locally at the developers' companies). This last scenario is particularly interesting, as it will allow software analysers from testing companies to perform code analysis tasks on encrypted software code, while preserving software privacy. The idea is that analysers should only access an encrypted version of the developer's code and should learn only the result of their code analysis task (e.g., if the code contains vulnerabilities) and nothing else about the code. In contrast, developers should be able to receive and understand the results of the analysis task and take action accordingly (e.g., remove the identified vulnerabilities).

Furthermore, with CoCoA and the proposed approach, this paper also initiates and defines a new research field – *Confidential Code Analysis* (CCA) –, which generalises the above-mentioned idea. CCA can open way for many new and interesting applications, such as different code analysis tasks and other approaches for vulnerability detection: type checking [74], code smell verification [55], and fault detection [8], [46], [54] are some examples, as well as integration into software development lifecycle and DevSecOps [59].

We formally analyse CoCoA's security and experimentally evaluate its effectiveness and performance when used to detect vulnerabilities in PHP web applications, the language used in 77% of such applications [68]. In our experiments, we use CoCoA to find SQLi and XSS vulnerabilities in two datasets: one with 3,205 programs from NIST SARD [52] and the other with 7 open-source applications. In both datasets, results show similar precision compared with non-confidential tools – 80% and 93% respectively (see Section 8 for details) –, and an average increase of 42.7% in performance costs, requiring only 209 ms to process 4k lines of code. We find this to be a modest overhead considering the benefits gained in privacy and functionality.

In summary, our main contributions are: (1) the definition of *Confidential Code Analysis* (CCA), a new field of research for *Software Quality and Privacy*, where static code analysis tasks are performed while preserving code's privacy and assuring its quality; (2) a first CCA solution based on SCA and SSE. The solution includes a data structure definition, encryption schemes, and an SSE-based static analysis algorithm for data flow analysis with control flow to represent the DCFG of a program's source code and ensure that SSE queries correctly follow the data dependencies of the code; (3) CoCoA, a prototype implementation of the solution, tailored for SQLi and XSS vulnerability detection in PHP web applications. This implementation is open source and available at [12]; (4) a security analysis and experimental evaluation of CoCoA which demonstrate its security, effectiveness, and performance, particularly when used for detecting web vulnerabilities in PHP code.

The remaining of the paper is organised as follows: Section 2 defines CCA. Section 3 presents an overview of CoCoA and its challenges, while Sections 4 and 5 detail CoCoA's design, Section 6 its security analysis, and Section 7 its implementation and deployment scenarios. Section 8 evaluates CoCoA in terms of theoretical cost analysis, effectiveness when applied to vulnerability detection, and performance and storage overheads. Section 9 shows how the tool can be extended for other scenarios. Lastly, Section 10 presents the related work, and Section 11 concludes the paper.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

2 Confidential Code Analysis

2.1 Formal Definition

We start by formally defining and analysing the concept of *Confidential Code Analysis* (CCA). Our intuition behind this concept is twofold: i) developers should be able to produce an encrypted representation of their code that does not reveal anything by itself, meaning that it can be securely shared with analysers or any other untrusted third party; ii) an analyser can submit a specification for an analysis task it wants to perform and receive (if authorized by the developer) a cryptographic token that, when combined with the encrypted code, outputs the result of the task.

We believe this intuition captures the goal of CCA. When a developer encrypts its code, this encryption does not reveal anything to adversaries, as in standard encryption. Nonetheless, at a later time, analysers can use this encryption to perform an analysis task of their choice. This step requires additional authorisation in the form of a cryptographic token that the developer must provide to allow the specified analysis. This puts the burden of scrutinising analysis requests on the developers, however it allows them to control who can analyse their code and which tasks they can perform. It also ensures that even if the encrypted code falls into the hands of an adversary, without access to one of these tokens it will not learn anything about the code. One could also think of other design choices, such as having the developer specify a privacy parameter and the system automatically determine which analysis requests to authorise based on that, but we leave the exploration of such ideas for future work.

The next definition captures the intuition described. Formally, a CCA scheme $\prod = (\mathsf{Encrypt}, \mathsf{Authorise}, \mathsf{Analyse})$ consists of three protocols that are executed between a developer and an analyser:

- $\{k, E_k(c)\} \leftarrow \prod.\mathsf{Encrypt}(\lambda, c)$ is executed by the developer and encrypts a software's source code, with inputs security parameter λ and source code c. At the end, the developer produces secret key k, which it keeps to itself, and an encrypted representation of the code $E_k(c)$, which can be shared with the analyser.
- $\{k_f\} \leftarrow \prod.\mathsf{Authorise}(k, f)$ is executed by the developer and creates an authorisation token for a specific analysis task provided by the analyser, with inputs secret key k and the task f. At the end of the protocol, the developer produces an authorisation token k_f, sharing it with the analyser.
- $\{f_{k_f}(E_k(c))\} \leftarrow \prod.\mathsf{Analyse}(f, k_f, E_k(c))$ is executed by the analyser and performs analysis task f by evaluating it on the encrypted code $E_k(c)$ through authorisation token k_f, i.e., $f(c) = f_{k_f}(E_k(c))$. At the end, the analyser produces the encrypted results of task f, which can be shared with the developer.

Regarding security, the intuition is that: $E_k(c)$ should not reveal anything about c (*code secrecy*); $f_{k_f}(E_k(c))$ should only reveal the result of task f when applied to c (*analysis secrecy with public result* – meaning that the analyser learns the result of the task (e.g., there is an XSS vulnerability) but nothing else about the code); and $f_{k_f}(E_k(c))$ should output the same result as $f(c)$ (*correctness*).

2.2 Adversary Model

We focus on securing *Software Code Analysis as a Service* application scenarios, where the developer's encrypted code is stored in a remote server (e.g., analysers' infrastructure or the cloud), and analysers access it to carry out their code analysis tasks. In this scenario, a semi-honest adaptive adversary is considered, meaning that it will execute the protocol as expected, but may try to break the confidentiality of the code being analysed or learn sensitive information about it that the developer did not explicitly authorise. The adversary may be the analyser itself or a third party impersonating it, or eavesdropping on their communications. We consider the developer to be trusted, as it is its source code that is being protected. We note that this adversary model is reminiscent of the one typically considered in cloud-based applications [2].

3 CoCoA Challenges and Overview

We aim to create a system that follows the CCA definition and so is capable of analysing encrypted code to discover vulnerabilities. We call it CoCoA and it combines *Static Code Analysis* (SCA) with *Searchable Symmetric Encryption* (SSE). Before introducing our approach, we present the challenges we faced in building such a CCA scheme.

3.1 Challenges

1) How to protect code privacy while supporting analysis tasks? The easiest way to protect the privacy of a program is to encrypt its source code. However, code analysis needs to understand the program's logic, semantics, and control flow, as well as the data dependencies between variables and function calls, which is impossible to do in encrypted code. Our idea is to represent the source code in a lightweight intermediate language that preserves these properties and, from there, extract its *Data and Control Flow Graph* (DCFG) and encrypt it with a scheme that allows computations over encrypted data. This will allow data flow analysis considering both data flow and control flow information.

2) Which cryptographic schemes can support DCFG analysis? There are many encryption schemes that support computations on encrypted data. Fully Homomorphic Encryption (FHE) [28] allows additions and multiplications of encrypted values. Since any computation can be expressed as an arithmetic circuit, FHE can be used as a general-purpose tool for encrypted data processing. Multi-Party Computation [13] also supports generic encrypted computations, although requiring multiple non-colluding parties. However, both schemes are still far from practical in terms of performance. Instead, we propose using SSE [27], a more practical solution designed for securely storing and searching encrypted text databases in remote untrusted servers. Our idea is to encrypt the DCFG with SSE, but with the necessary adjustments to support software code analysis.

3) How can we adapt SSE to encrypted software code analysis? Moving from keyword queries in encrypted databases to code analysis in encrypted code seems promising, but it is far from trivial. To accomplish this, many challenges arise, including: i) what type of data structure can represent the DCFG, i.e., the code and its data and control flows, for

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

encryption? ii) how to encrypt such a data structure so that it does not leak anything about the code when stored? iii) how to perform code analysis tasks through this encrypted data structure, revealing just the result of the intended computation? We will answer these questions in the next sections as we explain the details of our solution.

3.2 Overview

This section presents an overview of CoCoA, the solution that allowed us to solve the previous challenges. As we aim to preserve the privacy of the code while offering the ability to inspect it without compromising its secrecy, we propose an approach divided into two main phases – *Code Privacy* phase and *Encrypted Code Analysis* phase. Figure 1 shows CoCoA's architecture, with the first phase running on the Developer's side (i.e., in its PC/infrastructure), and the second phase running on the Analyser's side.

The *Code Privacy* phase (blue flow in the figure) prepares the code for encryption by extracting its DCFG, which captures the code's data and control flows, and representing it as an inverted index [15], [42], which will enable SSE queries on those flows, as a static data flow analysis would be carried out. Then, the index is encrypted following an SSE-like encryption scheme we designed, resulting in an *Encrypted Inverted Index* (EII) that can be securely stored in an untrusted location. This phase is executed on the Developer's side (i.e., in its PC/infrastructure), and it corresponds to CCA's Encrypt protocol.

The *Encrypted Code Analysis* phase (the orange and green flows in the figure) allows developers or authorised analysts to perform analysis tasks for vulnerability detection in the encrypted code, i.e., over the EII. The result of this phase is the vulnerable encrypted data flows, which are reported back to the developer for decryption and remediation of the vulnerabilities found. This phase corresponds to protocols Authorise and Analyse, thus involving both the developer and analyser, but its heavier computations are executed on the analyser's side.

Nonetheless, the solution is transparent for both sides, i.e., developers only need to submit their code and respond to Authorise requests, and analysers just select a *Code Analysis Task* (i.e., a vulnerability class to test), and CoCoA's modules perform the required computations and output the results. Sections 4 and 5 will detail these two main phases, respectively. Additionally, Figure 2 showcases CoCoA processing an example PHP program (in the left top corner) for an XSS vulnerability detection task, including the output of each module. The program contains an XSS vulnerability flowing through lines {1,8}.

4 Code Privacy Phase

This phase involves processing and preparing the application code through a five-stage pipeline that encrypts it while keeping data dependencies and control flow data. These stages are carried out by modules *ITL Converter*, *Data & Control Flows Extractor*, *Inverted Index Creator*, *Index Keys Generator* and *Encryptor*.

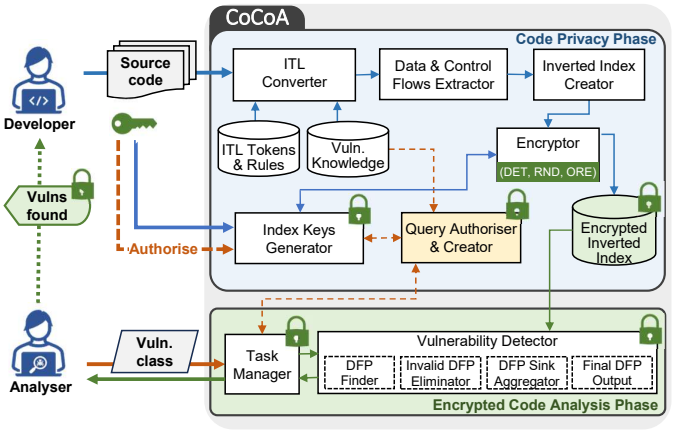

Fig. 1. CoCoA's architecture. ITL stands for *Intermediary Token Language*, DFP for *Data Flow Path*, and Vuln. for *Vulnerability*.

4.1 Intermediary Token Language (ITL) Converter

The application code is parsed and split according to the language specification (e.g., PHP) and its tokens, following, therefore, its syntactic and semantic aspects. The result is a LexToken stream composed of LexTokens [1], [34], where each one represents a broken part of the code, denoted by a token defined in the language specification, along with the code it represents and its position in the code. LexTokens are therefore expressed in the form of `<token_type, value, line_number>`. The type of LexTokens (`token_type`) can be, for example, `VAR` to represent variables, `FUNC_CALL` for function calls, `IF` and `FOR` for keywords of the language, and `LPAREN` for the left-parentheses metacharacter. Afterwards, LexTokens are normalised to remove irrelevant data (e.g., comments) and translated into an *Intermediate Token Language* (ITL), thus resulting in an *ITL-token* stream that preserves the logic, semantics, and data and control flows of the source code. The ITL must be simple, lightweight, and generic enough to enable static analysis and cryptographic techniques over it, while also abstracting/obfuscating the names of variables, operators, and user functions. To achieve this, the ITL Converter uses two databases – *ITL Tokens & Rules* and *Vuln. Knowledge* – during the ITL translation process (explained in § 4.1.2).

4.1.1 ITL-Tokens & Rules and Vuln. Knowledge Databases

These databases contain rules and code elements that are needed to convert the target programming language to ITL, as well as elements that can be interesting to search given the intended analysis tasks.

The first database defines ITL-tokens and establishes how to process and translate LexTokens to them (detailed in § 4.1.2). The tokens used in our ITL are mainly provided by the `token_type` of LexTokens, plus those specified for code analysis tasks we want to support (e.g., those presented in Table 1) and others that we define for specific purposes, such as denoting the end of assignments, conditions, and control flow branches. In addition, an ITL-token can represent several code elements with the same meaning (e.g., `$_POST` and `$_GET` entry points are translated into a single `INPUT` ITL-token).

The second database associates vulnerability-related ITL-tokens to their respective code elements from the target

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

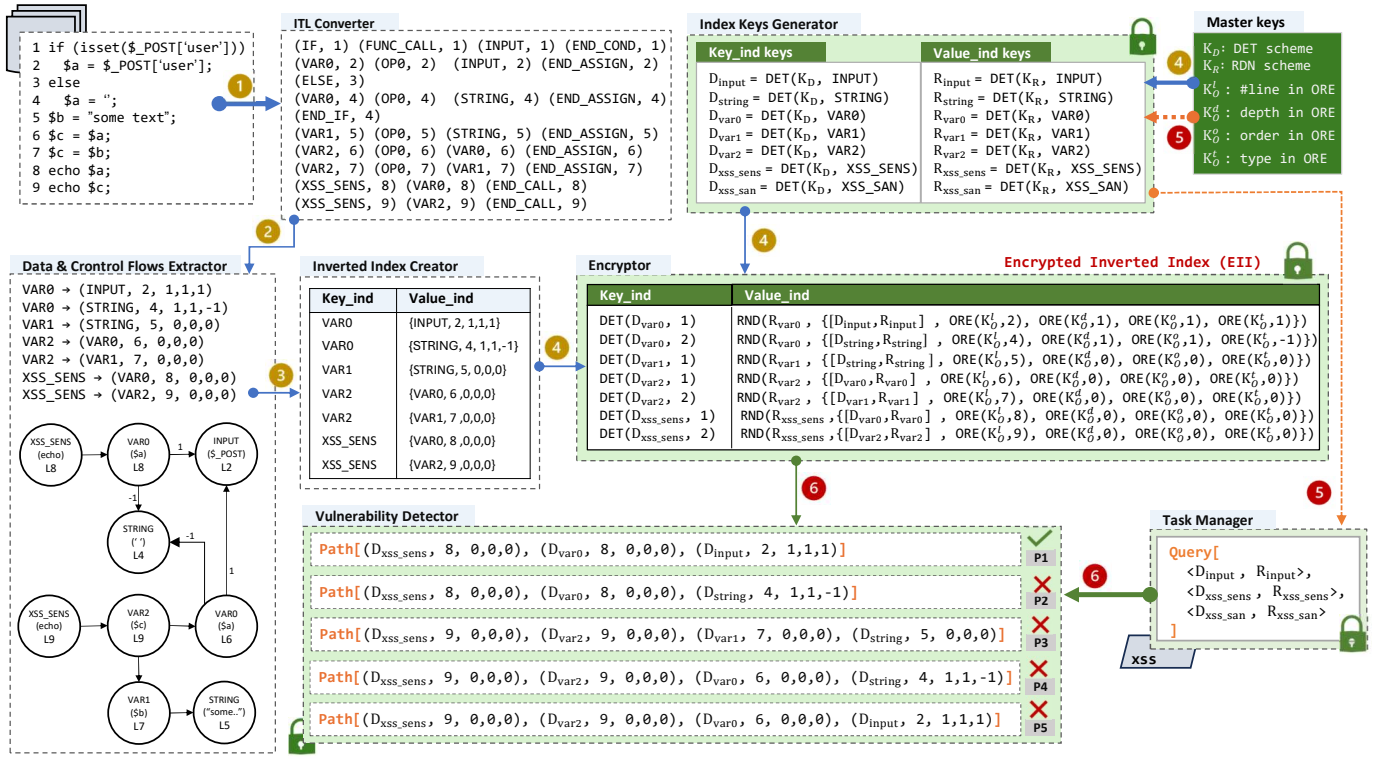

Fig. 2. Example of executing CoCoA with the XSS vulnerability detection analysis task in PHP. The encryption of numerical values in the Vulnerability Detector box is omitted for simplicity.

TABLE 1
Example of a Vulnerability (Vuln.) Knowledge database for detection of XSS and SQLi vulnerabilities in PHP.

ITL-Token	PHP code element
INPUT	$_GET, $_POST, $_FILES, $_SESSION, $_COOKIE
XSS_SENS	echo, print, exit
SQLi_SENS	mysqli_query, mysqli_stmt_execute, mysqli_execute
XSS_SAN	encodeForHTML, htmlentities, htmlspecialchars
SQLi_SAN	mysqli_real_escape_string, mysqli_stmt_bind_param

programming language. This can include entry points, sensitive functions, and sanitisation functions (functions that invalidate malicious inputs) which are then mapped to ITL-tokens related to specific vulnerability classes. Table 1 shows an example of this for the XSS and SQLi vulnerability classes in PHP. For XSS, for instance, the functions `echo` and `htmlentities` are, respectively, a sensitive sink and a sanitisation function, which are mapped to `XSS_SENS` and `XSS_SAN` ITL-tokens.

4.1.2 Translation process

Each group of LexTokens belonging to the same line of code is translated into an ITL line, after removing LexTokens of metacharacters that are irrelevant for representation as ITL and analysis (e.g., the left-parenthesis, `LPAREN`). After cleaning, each remaining LexToken of the code line is analysed and transformed into a tuple `<ITL-token, line_number>`, where the `ITL-token` will be either the LexToken's `token_type` or a specific token for the code analysis task (e.g., XSS detection), and the `line_number` is the line number presented in the LexToken. Afterwards, an *ending* ITL-token is added to the resulting ITL line, i.e., one of those we created to this effect, such as `END_ASSIGN` to denote the end of an assignment instruction.

Knowledge about specific analysis tasks: for each LexToken of the type `FUNC_CALL` or `VAR`, the translator matches its `value` with the data in the Vuln. Knowledge database, replacing such tokens with ones specific for the vulnerability classes considered. For instance, for PHP, the `VAR` LexToken is replaced by the `INPUT` ITL-token if its `value` starts with `$_`, thus indicating that it is an entry point. Also, the `FUNC_CALL` LexToken is replaced by an ITL-token from Table 1 if its `value` matches any function from there. For example, the `XSS_SENS` ITL-token is chosen if the LexToken's value matches the sensitive function `echo`.

Names of variables, user functions and operators: concrete token names for variables, user functions and operators are replaced by abstract ITL-tokens since, for analysis tasks, we only need to be able to distinguish different ITL-tokens, regardless of their concrete names. Such an abstraction also adds an extra layer of obfuscation and privacy. Thus, the token `VAR` (representing variables) is adapted to be different for each variable by appending an incremental counter to the end of the ITL-token. For example, in the code of Figure 2, both variables `$a` and `$b` are initially represented by the LexToken `VAR`; the translator determines that the ITL-token for the former will be `VAR0`, since `$a` appears first (line 2), while for the latter it will be `VAR1`. As the variable `$a` reappears later in the code (lines 4, 6 and 8), it will again be replaced by `VAR0` because the translator stores the context to track the variables that have already appeared. For user function names, the same procedure is applied, using the ITL-token `FUNC` followed by an incremental counter to distinguish the different user functions.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

Also, for operators, the translator follows the same process but with a slight difference: tokens representing different operators (e.g., `PLUS` for +, `EQUALS` for =) are replaced by a single ITL-token – `OP` – followed by an incremental counter, resulting in operators such as `OP0, OP1, OP2`.

The ITL Converter box of Figure 2 depicts the result of this translation process. We can observe that abstract ITL-tokens `VAR0` and `OP0` represent, respectively, the variable and operator, `INPUT` represents an entry point, and `END_ASSIGN` the *ending* of the instruction. Note that `FUNC_CALL` also appears, which denotes built-in functions that are not relevant for vulnerability detection purposes but are retained for semantic effects.

4.2 Data & Control Flows Extractor

Finding code interconnectivity over the ITL-token stream means finding the stream's existing data and control flows. While the former relates to data dependencies between ITL-tokens, the latter is associated with conditional ITL instructions. This module is responsible for obtaining both types of flows. While processing ITL-tokens, on the one hand, it creates connections of data dependencies and data flows throughout the code by analysing assignments and function calls, and, on the other hand, it extracts control flows by analysing conditional instructions. The resulting flows can be expressed as a graph, where the nodes are ITL-tokens, extended with the information needed to represent branched control flows, and the edges are connections. We call this graph the *Data and Control Flow Graph* (DCFG).

4.2.1 Data Dependency and Bottom-up Data Flow

Typically, data dependency is formed by connections, in which variables are used in assignment instructions and predicates throughout the program [24], [72]. We have extended this concept by also considering as connections variables that appear as parameters of function calls, since task analysis can also involve these cases.

As CoCoA's *Encrypted Code Analysis* phase relies on the reasoning behind SSE, i.e., an *inverted* index [15], the Data & Control Flows Extractor module must follow an inverted flow approach to build the data dependency graph, and thereby enable subsequent *bottom-up* data flow analysis. Therefore, data dependencies within assignment instructions will flow from left to right, e.g., for the `$a = $_POST['user'];` instruction, portrayed in ITL as `(VAR0,2)(OP0,2)(INPUT,2)(END_ASSIGN, 2)`, the module creates a connection pointing from `VAR0` to `INPUT`, i.e., `VAR0 -> INPUT` (meaning that `$a` points to `$_POST['user']`). We recall that CoCoA keeps the context of variables and operators that appear throughout the stream, and, therefore, the module can determine when an $\texttt{OP}_x$ ITL-token is an assignment operator in order to establish the connections correctly. In the case of function calls, data dependencies flow from the function name to its arguments, e.g., `XSS_SENS -> VAR0`. In Figure 2, at the bottom of the Data & Control Flows Extractor box, the DFCG is shown for the program we have been following. Observing the graph, we can see, for example, the dependencies of `VAR0` (`$a` variable) with lines 2, 4, 6 and 8.

```
1  if(1==1){
2    if(1==1)
3      $a = $_POST['u']; //[depth=2,order=1,type=1]
4    else
5      $a = $_POST['u']; //[depth=2,order=1,type=-1]
6
7    if(1==1)
8      echo $a; //[depth=2,order=2,type=1]
9  }
10 else
11   $a = $_GET['u']; //[depth=1,order=1,type=-1]
12 echo $a; //[depth=0,order=0,type=0]
```

Listing 1. A PHP program with various control flow branches and with different depths, orders and types, and their values.

4.2.2 Control Flow

Since static analysis does not evaluate the code while it is being executed, it cannot determine which branch of the program's control flow to follow. So, it needs to consider every branch that can occur, i.e., understand the program's control flow of conditional instructions (`if`, `switch` and `loops`) and whether or not one branch is independent of another. `switch` instructions are processed as being `if`, while `loops` are considered to be executed once and, hence, their instructions are processed as being outside the loops.

To this end, we extended the ITL tuples with three fields: `depth`, `order` and `type`, resulting in tuples in the form `<ITL-token, line_number, depth, order, type>`. To help explain the purpose of these fields, we present in Listing 1 a PHP excerpt with various control flow branches, where these fields are illustrated as comments. `depth` indicates the number of `if` statements nested inside another. For instance, the depth of ITL-tokens of line 3 is 2 because this line exists within a nested `if` statement of depth 1. Line 8 also has depth 2 for the same reason and is not nested inside the `if` of line 2. `order` denotes the number of `if` statements inside a depth. For instance, the order of any token in line 3 is 1 because it belongs to the first if-else block inside the `if` of line 1. Line 8 has order 2 since it belongs to the second `if` from the `if` of line 1. Finally, `type` is the type of the control flow, and its value describes whether the branch is an `if` (value 1), an `elseif` (a positive number that increments for each `elseif`), or an `else` (value -1). The type of line 3 is 1 because it is inside an `if`, but in line 5, it is -1 because it is inside an `else`. Yet, these fields are set to zero in line 12 since they are outside any control flow if-branch.

Combining the two approaches described above, the Data & Control Flows Extractor uses the ITL conditional instructions to fill these three fields and builds the DCFG by linking a series of pairs. In Figure 2, at the top of the Data & Control Flows Extractor box, these pairs are illustrated. For each pair, the left part indicates the ITL-tokens with dependencies, while the right part shows their ITL tuples dependencies, represented in their extended form. For example, the module for the second ITL instruction creates a connection pointing from `VAR0` to `INPUT`, i.e., the pair `VAR0 -> (INPUT, 2, 1,1,1)`, meaning that `$a` points to `$_POST['user']`, and that instruction is within a true `if`-branch. We can also observe that the second pair is the ITL instruction within the `else`-branch, and all the other

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

instructions that do not present control have their control flow fields set to 0. Note that ITL conditional statements are used only to extract control flow and thus to fill these three fields. So, they do not need to be represented in DCFG.

4.3 Inverted Index Creator

An *Inverted Index* [42] is a data structure that represents data in the form of <key_ind, value_ind> entries, where keywords (stored in key_ind) are mapped to the documents in which they appear in (stored in value_ind), with key_ind entries being unique so as to avoid ambiguities between keywords and the documents they map. An example application is unique keywords pointing to the IDs of text documents that contain them. The index is queried for keywords and returns the documents in which they appear. Each time a new keyword appears in a document, a new entry is added to the index, representing that occurrence and linking the keyword to the document.

After constructing the DCFG, the inverted index is built from there. Each DCFG pair will be an entry in the index, where the ITL-token (the left side of the pair) will be the key_ind and the extended ITL tuple (the right side of the pair) will be the index's value_ind. In the corresponding box of this module in Figure 2, we can observe the resulting inverted index from the DCFG.

4.4 Encryptor

Given the inverted index, the Encryptor module builds an encrypted version of it – the *Encrypted Inverted Index* (EII) – that will support code analysis tasks. This is achieved by adapting cryptographic techniques traditionally used for searching encrypted databases and text documents. We begin this section with a brief overview of these tecnhiques, then explain the approach used by the Encryptor module.

4.4.1 Searchable Symmetric Encryption (SSE)

SSE [65] is a cryptographic technique that allows a client to securely store and query a database in an untrusted remote server. In common among most approaches is the dependence on an inverted index [15], thus allowing queries to be performed in time sub-linear with the database size.

The index is usually encrypted using a combination of two encryption schemes, a deterministic (*DET*) and a probabilistic/randomised (*RND*). In a *DET* scheme, encrypting a message several times with the same cryptographic key will always produce the same ciphertext. Such a scheme is usually not considered secure since it leaks repetitions, unless the plaintext domain is unique (i.e., it holds no repetitions). Nonetheless, it supports an interesting feature: *equality-testing* in the encrypted domain, i.e., testing whether two DET encryptions are equal. In practice, there are many ways to build DET schemes, such as using a secure block cipher with fixed IVs or using a Hash-based Message Authentication Code (HMAC) (even though HMACs do not hold a decryption function, they can still be used to implement DET if decryption is not necessary). In contrast, an *RND* scheme will always produce different ciphertexts, even when encrypting a message multiple times with the same cryptographic key. This more secure scheme can be used to protect data, but it does not allow any computation in the encrypted domain. *RND* can be implemented by using a block cipher with random IVs, such as AES-CBC [38].

In SSE, key_ind is encrypted with *DET* while value_ind is encrypted with *RND*. Thus, the resulting EII index entries take the form <DET(K_D,key_ind), RND(K_R,value_ind)>, with K_D and K_R being cryptographic keys for *DET* and *RND* respectively. This allows leveraging the equality-testing feature of *DET* to search in the encrypted index while its encryptions reveal nothing, since by definition of inverted indexes, each key_ind is unique. As such, information leakage can only occur when a query is performed.

4.4.2 Order Revealing Encryption (ORE)

ORE [41] is a type of encryption scheme that preserves order relations between plaintexts, thus allowing order comparisons of numerical values to be performed after encryption. In ORE, a numerical value and a cryptographic key are taken as input, and a random byte string is produced as a result of encryption. Then, given two ORE encryptions, a comparison function outputs which encrypted value is higher, lower, or if equal, as if their plaintexts values were being compared. ORE is typically used in database systems that need to support range queries. In CoCoA we use ORE to encrypt line numbers and numerical control flow information (namely depth, order, and type) of the index's value_ind, so that these can be compared during code analysis without exposing their plaintext values.

4.4.3 Building the EII for Encrypted Code Analysis

To build the Encrypted Inverted Index (EII), CoCoA uses an approach similar to SSE, augmented with ORE so that no index data is ever exposed in plaintext, as described next:

- The entries of the inverted index will be represented as SSE index entries (i.e., EII entries). The unique nature of key_ind will be used to represent ITL-tokens, whereas value_ind will store the extended ITL tuples to which ITL-tokens are connected to;
- For each distinct ITL-token found in the inverted index, two cryptographic keys (one for *DET* and one for *RND*) will be generated. These keys will be used to encrypt key_ind and value_ind, respectively, and to replace ITL-tokens in the extended ITL tuples.
- The information stored in value_ind will allow traversing the EII, by linking to the next node of the DCFG graph (i.e., an EII entry) and storing the cryptographic keys needed to decrypt it.
- Numerical data of the value_ind (i.e., line numbers and control flow information) will be encrypted with *ORE* so that it can be used during code analysis without exposing its plaintext values;
- Each type of numerical data in the EII will be encrypted with a different ORE key, since we do not need to compare order relations between different types (e.g., line numbers are only compared between each other, and not with depth information).

The obtained EII can then be traversed as if it was the DCFG in its regular form, with index entries being decrypted as they are accessed but without ever exposing

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

any plaintext data from the source code. Next, we explain our solution in detail.

Cryptographic keys generation and usage.
CoCoA, through the *Index Keys Generator* module, employs six master keys: K_D and K_R to generate keys for the *DET* and *RND* schemes respectively, and $K_O^l, K_O^d, K_O^o, K_O^t$ to respectively encrypt line numbers, `depth`, `order` and `type` information with the ORE scheme.

Then, for each distinct ITL-token found in the DCFG, two additional cryptographic keys are generated by encrypting it with the *DET* scheme and keys K_D and K_R, respectively. For instance, in the Index Keys Generator box of Figure 2, two cryptographic keys for the ITL-token `VAR0` are generated by encrypting it with *DET* and master keys K_D and K_R, thus producing D_{var0}=DET(K_D,VAR0) and R_{var0}=DET(K_R,VAR0). These keys are then used to encrypt index entries related to `VAR0`, namely D_{var0} is used with *DET* to encrypt `VAR0`'s `key_ind` and R_{var0} with *RND* to encrypt `value_ind` connected to `VAR0`'s `key_ind`. In addition, they take the place of the ITL-token in the extended ITL tuple of `value_ind`, thus ensuring complete data secrecy while enabling EII traversal.

Composition of `key_ind` and ensuring its uniqueness.
Since we can have multiple assignments to the same variable and function calls of the same function in different code lines, we need a way to store this information in the index while guaranteeing that: i) each `key_ind` is still unique and ii) the index does not leak the frequency of each assignment and function call. Similarly to [7], we add a counter to each `key_ind` specific to the ITL-token, i.e., each token will have a unique counter. These counters start with value 1 and are incremented each time a specific ITL-token is found in the inverted index (i.e., on the left-side of the DCFG's pairs). The EII's `key_ind` is then calculated by encrypting the counter with the *DET* scheme and the cryptographic key associated with the ITL-token — DET(D_t, c_t), with D_t and c_t respectively being the cryptographic key and counter of the ITL-token. For instance, in Figure 2, in the Inverted Index Creator box, the `VAR0` token appears twice as `key_ind`, meaning that in the EII there are two `key_ind` entries using D_{var0} with *DET* (the first two rows of EII in the Encryptor box), one with the counter value of 1 (DET(D_{var0},1)) and another with the counter value of 2 (DET(D_{var0},2)). The same procedure is applied to `VAR2` and `XSS_SENS` tokens for the same reason. This approach guarantees that: all `key_ind` depend on their respective ITL-tokens; they are always unique; and they ensure the privacy of the ITL-token since it is never exposed in plaintext.

Composition of `value_ind` and ensuring EII traverse.
We need to build `value_ind` in a way that allows us to traverse and decrypt the EII as we perform data flow analysis tasks, but only entries relevant to a task should be decryptable and the information from the source code should never be exposed in plaintext. Additionally, and in contrast to traditional SSE, here we are not querying for keywords, but for data dependencies and control flows. Hence, we need to not only create entries for each ITL-token, but also establish the respective connections between them.

Inspired from [15], we use `value_ind` to point to the next EII entry from the data flow and store the cryptographic keys needed to access and decrypt it. More specifically, for each index entry, we replace the ITL-token of the extended ITL tuple with its corresponding cryptographic keys and encrypt the whole tuple with the *RND* scheme. Example: consider the index entry <XSS_SENS, {VAR0,8,0,0,0}> from Figure 2 (line 6 of the Inverted Index Creator box) and its encryption as presented in the Encryptor box. Its EII `value_ind` contains D_{var0} and R_{var0}, which are the cryptographic keys needed to access and decrypt the next entries of the data flow (i.e., lines 1 and 2 in the same box). These cryptographic keys represent the ITL-token `VAR0`, replacing it in the EII extended ITL tuple of line 6. Then, the whole tuple is encrypted with *RND* and key R_{xss_sens}, thus obtaining RND(R_{xss_sens}, {D_{var0},R_{var0}, ...}).

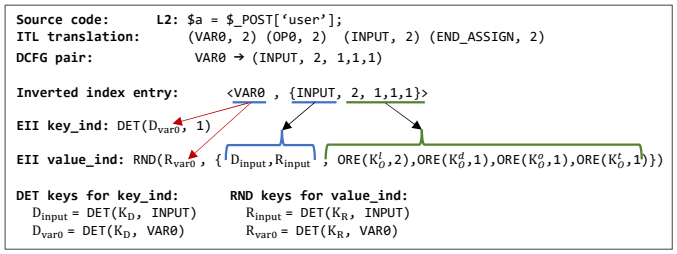

Fig. 3. Example of the transformation of an instruction of code into an EII entry.

Keeping numeric values secret in `value_ind`.
To ensure full code privacy and no information leakage from decrypting `value_ind` during code analysis, our approach also encrypts the line number and control flow information contained in each extended ITL tuple (i.e., `depth`, `order` and `type` fields). Since this data is only used in order comparisons, the four fields are encrypted using *ORE*. Additionally, since these fields do not need to be compared with each other, each can be encrypted using a different master key – $K_O^l, K_O^d, K_O^o, K_O^t$ –, as described previously. In the index entry <VAR0, {INPUT,2,1,1,1}> of our example, the four numeric values of `INPUT` will be encrypted as follows: (ORE(K_O^l,2),ORE(K_O^d,1),ORE(K_O^o,1),ORE(K_O^t,1)).

The final form of EII entries.
The next formula summarises the transformation from a <DCFG$_t$, DCFG$_{t+1}$> pair to an EII entry of the form <key_ind, value_ind>, where t is an ITL-token, $t+1$ is the next token in the stream (i.e., DCFG$_t$ points to DCFG$_{t+1}$), and c is a counter. Additionally, Figure 3 illustrates the complete transformation of the instruction `$a = $_POST['user']` (line 2 from Figure 2) into an EII entry.

$$\begin{aligned}&\langle DCFG_t, DCFG_{t+1}\rangle \Rightarrow \langle DET(D_t, c_t),\ RND(R_t, EII_{t+1})\rangle :\\ &D_t = DET(K_D, t),\ c_t = \{1, \ldots\},\ R_t = DET(K_R, t),\\ &EII_{t+1} = \{D_{t+1}, R_{t+1}, ORE(K_O^l, line_n), ORE(K_O^d, depth),\\ &\qquad ORE(K_O^o, order), ORE(K_O^t, type)\}\end{aligned}$$

5 Encrypted Code Analysis Phase

At the end of the code privacy phase (CCA protocol Encrypt), the encrypted inverted index (EII) is obtained, and the developer can give the analyser access to it, i.e., to themselves or someone else from their software company, or even external software testers from third-party testing companies. To perform code analysis tasks (CCA protocol Analyse), the analyser chooses the tasks it wants to perform and asks

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

the developer to provide the cryptographic keys of the ITL-tokens (for *DET* and *RND* schemes) that are associated with those tasks (protocol Authorise). This is an important step, as it gives the developer an extra layer of access control over who can perform analysis tasks and which tasks they can perform. Given those keys, the analyser starts accessing the encrypted index, following the relevant data flows and decrypting index entries as it goes, and in the end gets the results of the tasks. In more detail, and as shown in Figure 1, this phase is carried out by the collaboration of modules *Task Manager* and *Vulnerability Detector* on the analyser side, and *Query Authoriser & Creator* on the developer side.

5.1 Task Manager

This module receives an *Analysis Task* chosen by the analyser, such as the XSS vulnerability class for finding this type of vulnerability in the developer's code. Then the module interacts with the developer's *Query Authoriser & Creator* module, to ask for the developer's authorisation and create the query subjacent to the analysis task. If authorised, the *Vuln. Knowledge* database is accessed to retrieve the information previously defined for that task, namely the ITL-tokens associated with it (i.e., the ones representing entry points, sensitive and sanitisation functions of that vulnerability class), and then the cryptographic keys of the selected ITL-tokens are generated (through the Index Keys Generator module), thus composing the query with these pairs of keys. For instance, for the *XSS* analysis task, the ITL-tokens would be `INPUT`, `XSS_SENS`, and `XSS_SAN`, and the resulting query would be `Query[<`D_{input}`,`R_{input}`>, <`D_{xss_sens}`,`R_{xss_sens}`>, <`D_{xss_san}`,`R_{xss_san}`>]`. Lastly, the module sends the query to the *Vulnerability Detector* module to proceed with the analysis task, outputting its final result to the analyser. Formally, the steps taken to produce an authorised query implement CCA protocol Authorise.

5.2 Vulnerability Detector

This module implements protocol Analyse. Specifically, it takes the cryptographic keys for querying the index, extracts the code paths for the *Analysis Task*, and checks them for the vulnerability class, following the next four-step algorithm.

1) Finding data flow paths (DFP). This step starts by accessing the EII index with the cryptographic keys, following a bottom-up data flow approach. We recall that due to the nature of the inverted index, in which function call flows must be represented in an inverted way, and so their names appear as `key_ind`, we must follow a bottom-up approach. This means that we start the analysis by the end-point (last flow) of a vulnerability search – i.e., a sensitive sink. Hence, in our example, we start the analysis with D_{xss_sens} and R_{xss_sens} keys, as they mark the last flow of an XSS vulnerability. However, remember that due to possibly repeated assignments and function calls, `key_ind` is composed not only of the encrypted token D_t but also of its counter c_t. This means that to start accessing the index, the Vulnerability Detector must combine D_t and c_t through the *DET* scheme, as was previously done by the Encryptor to compose `key_ind` and ensure `key_ind` uniqueness. Moreover, as neither the vulnerability detector module nor the analyser has any knowledge about the EII index content, and thereby does not know how often a given token is repeated, the module will start with c_t=1 and check if the resulting ciphertext (i.e., `DET(`D_t`,1)`) exists as a EII's `key_ind`, continuing with increasing counter values until one fails to be found. For instance, in the example of Figure 2, D_{xss_sens} appears twice (in the Encryptor box), meaning that the module will create and test `DET(`D_{xss_sens}`,1)`, `DET(`D_{xss_sens}`,2)`, and `DET(`D_{xss_sens}`,3)`, stopping with the last one as it will not be found in the index. For each `key_ind` entry that is found, its corresponding `value_ind` is decrypted with R_t, and the resulting data flows are followed until their end is reached (i.e., a terminal ITL-token like `INPUT`), repeating the same procedure. The result is a list of encrypted data flow paths (DFP) that will be processed by the next steps of the algorithm to determine whether they are vulnerable. Each DFP starts with an ITL-token from a `key_ind`, followed by one or more extended ITL tuples provided by the corresponding `value_ind`. Figure 2 shows this list with five DFP paths ($P1$ to $P5$), in the Vulnerability Detector box. For a better understanding and clarity of the tuples contained in the encrypted DFPs, we present their numerical data without the ORE encryption. But we recall that these DFPs are managed by the module and presented to the analyser completely encrypted, revealing nothing about their content.

2) Removing invalid DFPs. Next, the algorithm filters out invalid paths, such as those with tokens whose `line_number` is greater than the `line_number` of the DFP's first token but belong to the same scope (e.g., code block, user function). For example, suppose that on the path $P3$, the token `(`D_{string}`,5,0,0,0)` had its `line_number` equal to 12, which is greater than the `line_number` of the first token `(`D_{xss_sens}`,9,0,0,0)`. $P3$ would be an invalid path because line 12 cannot be executed before line 9. Hence, these kinds of DFPs can be safely discarded, as they clearly do not exist in the context of the program.

3) Aggregating DFPs. As we follow a bottom-up approach, the starting DFP's token is the `final_token` associated with the code analysis task. So, this step groups and sorts DFPs by this final token and its line number, i.e., `<final_token, line_number>`, thus creating sub-list of such pairs. For the D_{xss_sens} final token, in our example, the sub-lists would be [P1, P2] for `<`D_{xss_sens}`,8>` and [P3, P4, P5] for `<`D_{xss_sens}`,9>`.

4) Outputting the final DFPs. This step determines which paths are correct, considering the program's control flow and whenever `depth` is different from zero. It employs three analyses over each sub-list of DFPs resulting from step 3): (i) resolving control flow branches, (ii) obtaining the closest DFP to the `final_token`, and (iii) determining whether a path is vulnerable.

In (i), for each distinct `depth`, paths are separated into sets: one with all paths with different `orders` regardless of `type`, and others including paths with equal `order` but different `type` (one set for each `order`). The first set will contain paths in which variables are propagated between `if`s, while the other sets will contain paths of the same `if`. Based on Listing 1 and the sink of line 8, examples of the first case are the paths `(2,2,1)`, `(2,1,-1)` (lines 8 and 5) and `(2,2,1)`, `(2,1,1)` (lines 8 and 3). Examples of the second case are, in our example, the paths [P1, P2] and [P4, P5]. Therefore, after applying this step, our initial sub-lists

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

are organised as follows [P1,P2] and [P3, [P4,P5]].

In (ii), for the resulting sets of a `depth` and for the paths whose tokens have these three fields equal, excluding the `final_token` or also excluding the last token of the path, the algorithm finds the closest path to the final token. In other words, it looks for the first difference in the paths and checks which one is the closest to the `final_token`. For example, supposing that P1 and P2 have equal control flow, including their last token, i.e., `(1,1,1)`, both paths would have these three fields equal, and the first difference between them would be the `line_number` of their last token, hence P2 would be chosen because its `line_number` is the one closest to $(D_{xss_sens}, 8, 0, 0, 0)$. In our example, as this scenario is not observable in both sub-lists from (i), they follow as such to step (iii).

In (iii), the resulting paths are checked for the vulnerability class subjacent to the analysis task, relying on three determinations: (a) for the resulting DFPs of (ii), all paths ending with D_{input} token are chosen and checked which ones do not contain a sanitisation token (e.g., D_{xss_san} `for XSS`); (b) for the resulting sub-sets of DFPs of (i) that did not fall in (ii) and in which the `type` field of their last tuple differs, the paths ending with D_{input} token are chosen and checked which ones do not contain a sanitisation token. In our example, sub-sets [P1,P2] and [P4,P5] fall into this scenario. P2 and P4 are discarded, as their last tuple does not end with D_{input}. So, from [P1,P2] and [P3,[P4,P5]] sub-lists came from (i), results [P1] and [P3,P5]; (c) for the resulting paths of (a) and (b) within sub-lists of the `final_token`, the step (ii) is applied, followed by the filter (a).

Following our example, at this point, the sub-list from $<D_{xss_sens}, 8>$ is [P1] and the sub-list from $<D_{xss_sens}, 9>$ is [P3, P5]. Applying step (ii) and filter (a) to this second sub-list results in no paths, meaning therefore that both paths are dropped. Despite P3 results from (ii) (thus dropping P5), it is discarded by (a) because it does not end with D_{input} token. The sub-list containing only [P1] is checked for the presence of the D_{input} token as its last token and for the absence of sanitisation tokens. $P1$ satisfies both conditions, therefore considered vulnerable, and thus it is the final result of verification (iii) and of the whole vulnerability detection process.

Lastly, the vulnerable DFPs are outputted by the Vulnerability Detector, which returns them to the Task Manager and subsequently to the analyser. The analyser can then share the results with the developer and provide suggestions on improving the source code based on the vulnerability class it chooses to analyse. In our example of Figure 2, only $P1$ is outputted as an XSS vulnerability. In this case, the analyser may suggest using the sanitisation function `htmlentities` to remove vulnerabilities of XSS.

6 Security Analysis

Our goal with CoCoA is to securely implement the functionality idealised for *Confidential Code Analysis* (CCA). This means that: (i) its index should not reveal anything about the code; (ii) and its query tokens, when combined with the index, should only reveal the result of the intended analysis task. Formally, we will use the leakage function $\mathcal{L}^{\text{CoCoA}} = (\mathcal{L}^{\text{CoCoA}}_{Encrypt}, \mathcal{L}^{\text{CoCoA}}_{Authorise}, \mathcal{L}^{\text{CoCoA}}_{Analyse})$ to capture this information and analyse CoCoA's security.

Demonstrating (i), we have an encrypted index (the EII) composed of multiple <`key_ind`, `value_ind`> entries. `value_ind` encryptions will not reveal anything since they are produced by *RND* and all have the same size before and after encryption. `key_ind` encryptions also do not reveal anything, since even though encrypted with *DET*, each represents either a different ITL-token or the same token but different counter value. Hence, the only thing leaked by the EII is its size, which represents the total number of DCFG's pairs. We deem this as basic information, nonetheless it could also be hidden through padding with dummy index entries. This means that $\mathcal{L}^{\text{CoCoA}}_{Encrypt} = \perp$.

Demonstrating (ii), we must first analyse the leakage of query tokens by themselves and then when combined with EII. For the first part, query tokens are encrypted with *DET*, which leaks repetitions. This means that when invoking the Authorise protocol, the adversary learns the deterministic encrypted query tokens associated with a task of its choice. Formally, this must be modelled as $\mathcal{L}^{\text{CoCoA}}_{Authorise} = \{\{D_0, R_0\}, \ldots, \{D_k, \ldots, R_k\}\}$, where $\{D_i, R_i\}$ is an encrypted token pair associated with the query.

For the second part of (ii), we must analyse the knowledge gained by the adversary when accessing the index with an encrypted query token. Indeed when an analysis task is performed, the adversary learns $\{P_0, ..., P_n\}$, the set of paths accessed when solving the query, with $P_i =< T_0, ..., T_n >$ being a path with multiple index entries $T_j =<D_j, R_j, ORE^l_j, ORE^d_j, ORE^o_j, ORE^t_j>$ that were accessed, where $\{D_j, R_j\}$ is the *DET* encrypted token pair associated with the entry, ORE^l_j is the *ORE* encryption of its line number, ORE^d_j of its `depth`, and so on. Formally, this is modelled as $\mathcal{L}^{\text{CoCoA}}_{Analyse} = \{P_0, ..., P_n\}$.

Although $\mathcal{L}^{\text{CoCoA}}_{Analyse}$ seems to be leaking many different things, this is either basic information or is part of the intended computation and, hence, unavoidable. $\{P_0, ..., P_n\}$ represents the found paths. We deem this as part of the result of the intended analysis task. Moreover, the adversary only learns the number of paths, their sizes, the *DET* encrypted tokens in each, and the corresponding *ORE* encryptions. *DET* encrypted tokens reveal co-occurrence of ITL-tokens between paths but are still encrypted and even if their encryptions would be broken, ITL-tokens have been obfuscated before encryption by replacing operators and variable/function names with abstract tokens. *ORE* encrypted line number and control flow information reveal frequency and order relations, but only between each field (i.e., the adversary can compare ORE^l_j with ORE^l_i but not with ORE^d_j) since each is encrypted with a different key.

7 CoCoA Implementation & Deployment

7.1 Implementation

We implemented a prototype of CoCoA [12] in Python to process PHP applications for XSS and SQLi vulnerability detection tasks. All modules were developed from scratch, with the exception of the Lexer sub-module, which was based on PLY's Lex module [5]. Our ITL totals 40 distinct ITL-tokens: 23 from LexTokens, 12 for *ending* instructions

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

(e.g., assignments, conditionals), and 5 for XSS and SQLi vulnerabilities. Our current prototype covers almost the whole PHP specification, except for some less commonly used instructions (e.g., tri-dimensional or greater arrays) and some object-oriented instructions. Also, although it supports analysing user function calls, it still needs improvements at the inter-procedural level regarding files. To implement the cryptographic operations, we used SHA1-HMAC for the *DET* scheme and AES-128 in CBC mode for *RND*, through the *Hashlib*, *Hmac*, and *PyCryptodome* Python libraries. For *ORE*, we used the ORE scheme by Lewi and Wu [41] and their *FastORE* library implementation [23], with 32-bit input size and 8-bit blocks. Lastly, databases were built with YAML [73]. CoCoA can be extended to other code analysis tasks and programming languages – Section 9 gives an overview of how this can be done.

7.2 Deployment Scenarios

CoCoA's approach was designed to ensure complete code privacy while enabling the analysis of the encrypted code by authorised software analysts. As previously stated, we envision three scenarios in which CoCoA can be deployed and used (as illustrated in Figure 4):

Scenario 1. The tool is deployed on the developer's side, at the development software company, and is used only to protect the code and the developer's intellectual property (IP), i.e., encrypted code analysis is disabled. Even though this scenario is possible, it does not explore the full capacities of CoCoA;

Scenario 2. It is an extension of Scenario 1, in which CoCoA is totally deployed and used on the developer side, allowing code analysis tasks over EII by analysers that could be developers themselves or the security team (e.g., purple team) existing in the software company.

Scenario 3. CoCoA is used as a service (CoCoAaaS) by software testing companies, in which software companies contract the testing software service to external companies and authorise them to analyse their code in a protected manner. The deployment is bi-parted between the developer (software company) and the analyser (the testing company). CoCoAaaS is deployed locally within the infrastructure's testing company or in the cloud, both comprising the EII index and the modules Task Manager and Vulnerability Detector, thus allowing code analysis tasks by the analyser. On the other hand, the remaining CoCoA modules are deployed on the developer side, giving them control over their code and the tasks that can be performed over EII. This scenario is particularly interesting because, besides preserving the privacy of the code, protecting the developer's IP and allowing code analysis tasks by external analysts in a controlled manner, it also protects the testing companies against possible defamations of non-disclosure agreements (NDA) regarding the code, and allows them to perform the tasks remotely, i.e., out-of-house, instead of the standard in-house (of the software company).

8 Evaluation

We divide our evaluation section between theoretical analysis and experimental evaluation.

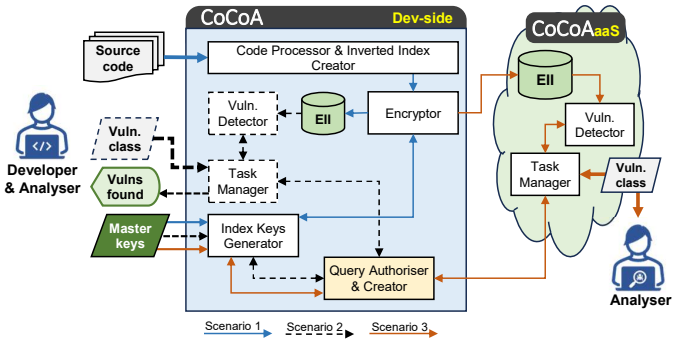

Fig. 4. Deployment scenarios of CoCoA.

8.1 Theoretical Analysis

In this section, we perform a complexity analysis of CoCoA regarding both communication and storage overheads. In CoCoA, we have three protocols: Encrypt, Authorise, and Analyse. As such, we evaluate each protocol separately.

Regarding Encrypt, this protocol is executed solely on the developer's side. Nonetheless, the resulting EII can be shared with the analyser when the protocol finishes. As such, we consider a communication complexity of $O(n)$, where n is the size of the EII in terms of the number of data flows and dependencies extracted from the source code. As for storage complexity, the developer has to store six master keys of constant size (O(1)), and either it or the analyser stores the EII. Hence, the storage complexity is $O(1+n) = O(n)$.

For the Authorise protocol, the analyser sends an analysis task to the developer (which we can assume to be of constant size, e.g., "XSS") and in return receives the cryptographic keys of the associated ITL-tokens if authorised, or nothing if not. Hence, the communication complexity of this step is $O(t)$, where t is the number of ITL-tokens associated with the task. In terms of storage complexity, we assume 0 (zero) since the exchanged data is temporary, only needed to perform the task, and can be discarded later.

Finally, for the Analyse protocol, this step is performed mostly by the analyser, but the final results can be shared with the developer. As such, we consider a communication complexity of $O(r)$, where r is the size of the result set of the analysis task (e.g., the number of vulnerable paths for the XSS analysis task). In terms of storage, we again consider a complexity of 0 (zero), since the resulting set is temporary and can be discarded after the developer fixes its code.

Concluding, CoCoA's overall communication complexity is $O(n+t+r) = O(n)$ (since t and r must forcibly be a subset of n) and its storage complexity is $O(n)$.

8.2 Experimental Evaluation

The objective of the experimental evaluation was to answer the following questions: i) can CoCoA perform code analysis with high precision and a low false positive rate? ii) how do CoCoA's results fare compared to standard (non-confidential) static analysis tools? iii) what are the performance and storage overheads imposed by our approach?

8.2.1 *Vulnerability Detection Effectiveness*

This section evaluates CoCoA's effectiveness in detecting XSS and SQLi vulnerabilities. For this, we built and used

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

TABLE 2
Metrics used to evaluate the tools in vulnerability detection

Metric	Definition and Formula
Accuracy *(acc)*	Measures how correctly the tool classifies instances as vulnerable (TP - True Positive) and not-vulnerable (TN - True Negative) over the total number of instances. $acc = (TP + TN)/(TP + FN + TN + FP)$
Recall *(rec)*	Measures the proportion of actual vulnerabilities that are correctly identified by the tool. $rec = TP/(TP + FN)$
Precision *(pr)*	Measures the proportion of correctly identified vulnerabilities out of all instances flagged as vulnerabilities by the solution. $pr = TP/(TP + FP)$
F1-score	Measures the harmonic mean between *pr* and *rec*, providing a single metric that balances the trade-off between the two. $F1 = 2 * (pr * rec)/(pr + rec)$
FP rate *(fpr)*	Proportion of not-vulnerable (FP- False Positive) cases incorrectly classified as vulnerable. $fpr = FP/(FP + TN)$
FN rate *(fnr)*	Proportion of vulnerable cases missed (FN - False Negative) by the tool. $fnr = FN/(FN + TP)$

two PHP application datasets – one synthetic and one real-world – and compared their results with 3 standard, non-confidential static analysis tools: Pixy [36], PHPCorrector [48], and WAP [45]. The datasets were made public and are included with CoCoA's source code [12]. Additionally, the metrics used to evaluate the tools are defined in Table 2.

Detection with Synthetic Applications. We built a synthetic ground-truth (GT) dataset comprising 3,205 SQLi and XSS PHP code excerpts provided by SARD [52]. This GT includes a diversity of vulnerable (Vul) and not-vulnerable (N-Vul) cases with varying coding forms. These range from the standard use of sanitisation functions (e.g., `mysqli_real_escape_string`) and appearance of vulnerabilities in the code to uncommon forms of validating user inputs (e.g., `filter_var` and casting to data types). GT also includes object-oriented code and alternative methods for reading input data (e.g., `fgets`) besides the superglobal variables (e.g., `$_GET`, `$_POST`). The dataset is balanced with 1,569 Vul and 1,636 N-Vul excerpts, containing 1,441 SQLi code excerpts (861 Vul and 580 N-Vul) and 1,764 XSS code excerpts (708 Vul and 1,056 N-Vul).

We executed CoCoA and the 3 tools with this dataset. Both CoCoA and WAP processed all cases without errors. Pixy and PHPCorrector had trouble with 477 and 148 cases, respectively, due to the handling of object-oriented code and conditional instructions involving function calls. Table 3 shows the results, expressed using the metrics in Table 2, calculated from the confusion matrices. Results are presented for each task in separate (XSS, SQLi) and combined (GT).

CoCoA presented similar precision (*pr*) to that of PHPCorrector, and both outperformed Pixy and WAP. PHPCorrector achieved the highest *pr* but also the lowest recall (*rec*). Pixy achieved the highest *rec*, but also showed a higher false positive rate (*fpr*), which can be counterproductive for this type of tools. In contrast, CoCoA's *fpr* was much lower, and its *rec* and F1-score aligned with WAP. We believe these lower results are not due to CoCoA's design but to limitations in our current prototype implementation, especially regarding inter-procedural analysis and object-oriented instructions, and thus can still be improved.

TABLE 3
Evaluation CoCoA and competitors with the GT dataset.

	CoCoA			Pixy			PHPCorrector			WAP		
Metric	**XSS**	**SQLi**	**GT**	**XSS**	**SQLi**	**GT**	**XSS**	**SQLi**	**GT**	**XSS**	**SQLi**	**GT**
Precision (pr)	0.70	0.97	0.80	0.54	0.74	0.64	1.00	0.73	0.86	0.35	0.65	0.39
Recall (rec)	0.30	0.19	0.24	0.61	0.87	0.74	0.02	0.02	0.02	0.50	0.12	0.29
Accuracy (acc)	0.67	0.51	0.60	0.64	0.71	0.66	0.61	0.42	0.53	0.42	0.43	0.43
F1-score	0.42	0.32	0.37	0.57	0.80	0.68	0.04	0.03	0.03	0.41	0.20	0.33
FP rate (fpr)	0.10	0.00	0.06	0.35	0.60	0.41	0.00	0.01	0.01	0.63	0.09	0.44
FN rate (fnr)	0.70	0.81	0.76	0.39	0.13	0.27	0.98	0.99	0.98	0.50	0.89	0.71

Detection with Real Applications. We further assessed CoCoA's effectiveness by analysing seven open-source PHP real web applications with known vulnerabilities, selected from [45], and comparing its results with the 3 tools. Table 4 shows the results obtained. The XSS and SQLi columns represent the number of vulnerabilities correctly found by the tools, while FP and FN columns show the number of false positives and false negatives, respectively. The last five columns of the table show the metrics calculated from the confusion matrix of the tools, expressing, thus, further analysis of these results.

CoCoA had the best *pr* – 93% –, like WAP, and outperformed the other tools in the *fpr* – 18% –, where it only wrongly detected 4 data flow paths as vulnerable. WAP was the best tool, but it yielded results similar to CoCoA. CoCoA and WAP correctly detected a similar number of vulnerabilities: 55 and 66; hence, CoCoA had less *rec* than WAP (56% and 64% respectively), meaning it missed a few more vulnerabilities. The *F1-score* is 70% for CoCoA versus 76% for WAP. Overall, PHPCorrector exhibited the worst results in almost all metrics. It only outperformed Pixy in *pr* (83% and 72% respectively). Pixy was only capable of processing 4 out of the 7 applications, so only these were considered for the metrics.

We also tested CoCoA without encryption (i.e., by disabling the Encryptor module). The results were the same as with encryption and hence omitted for simplicity. This shows that our encryption approach does not affect analysis effectiveness.

Comparing the results obtained with the two datasets and taking the *F1-score* metric as a reference, as it considers pr and rec and thus gives a good idea of a tool's effectiveness, CoCoA kept its second-best tool but doubled its *F1-score* (70%), probably justified by the absence of unusual coding forms and object-oriented code in some of these real applications. Contrarily, when we analyse its fpr, it increased from 6% to 18%, which means that it wrongly classified more data flow paths as vulnerable. This was due to inter-procedural analysis between files not being as precise in our current prototype. We recall that each instance of GT is a single file, whereas real applications comprise more than one file. So, this increment in fpr was expected.

8.2.2 Performance Overhead

To evaluate performance, we used the real dataset and conducted a series of tests with the XSS task to measure the execution time of each of CoCoA's modules. Each test was repeated and measured five times, and their average values were obtained and presented in Table 5. For the tests, we used a laptop with an AMD Ryzen 5 5600H CPU at 3.3GHz, 16Gb of RAM, and 512GB SSD, running Manjaro Linux 23.1.

Preprocessor. The preprocessor (Prep in the table) is a module that does not figure in CoCoA's design, but was added to our prototype to filter non-PHP code and non-PHP files from the dataset. Depending on the web application, this module can take from 1 ms to 17 ms to execute. After its

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

TABLE 4
Results and metrics of CoCoA, Pixy, PHPCorrector and WAP with real web applications.

	CoCoA				Pixy				PHPCorrector				WAP								
WebApp	**XSS**	**SQLi**	**FP**	**FN**	**XSS**	**SQLi**	**FP**	**FN**	**XSS**	**SQLi**	**FP**	**FN**	**XSS**	**SQLi**	**FP**	**FN**	**Metric**	**CoCoA**	**Pixy**	**PHPCorrector**	**WAP**
CurrentCost [14]	0	3	0	5	5	3	3	0	0	0	0	8	4	3	2	1	**pr**	0.93	0.72	0.83	0.93
DVWA [21]	13	7	2	0	0	4	2	16	2	0	0	18	2	4	2	14	**rec**	0.56	0.63	0.36	0.64
Ghost [30]	8	1	0	1	–	–	–	–	5	1	0	4	5	2	0	3	**acc**	0.61	0.56	0.42	0.67
Peruggia [56]	2	1	0	17	–	–	–	–	5	10	2	5	3	15	0	2	**F1-score**	0.70	0.67	0.50	0.76
Samate [52]	15	3	2	1	11	4	1	4	4	0	0	15	11	3	0	5	**fpr**	0.18	0.64	0.32	0.23
WackoPicko [69]	2	0	0	9	–	–	–	–	8	0	5	3	5	3	0	3	**fnr**	0.44	0.37	0.64	0.36
Zipec [75]	0	0	0	10	7	2	8	1	0	0	0	10	0	3	1	7					
Total	**40**	**15**	**4**	**43**	**23**	**13**	**14**	**21**	**24**	**11**	**7**	**63**	**30**	**33**	**5**	**35**					

TABLE 5
Execution time in milliseconds (ms) of the CoCoA modules, and comparison with WAP, for the XSS task, including encryption overheads.

			Prep	**Lexer**	**Trans**	**DCF and Encryptor**			**Code Privacy Phase Total**					**Vuln. Detector**			**Total Execution (CP+VD)**					**WAP**
WebApp	**F**	**LoC**	**Exec time**	**Exec time**	**Exec time**	**DCF -Enc**	**DCF +Enc**	**DCF +ORE**	**CP -Enc**	**CP +Enc**	**Enc OH (%)**	**CP +ORE**	**ORE OH (%)**	**VD -Enc**	**VD +Enc**	**VD +ORE**	**Total -Enc**	**Total +Enc**	**Enc OH (%)**	**Total +ORE**	**ORE OH (%)**	**Exec time**
CurrentCost	2	60	1.21	0.47	1.23	0.38	1.44	9.78	3.3	4.35	32.16	12.69	285.15	0.02	0.04	0.04	3.31	4.39	32.57	12.73	284.26	113
DVWA	8	291	1.50	2.37	5.52	1.55	5.98	45.08	10.94	15.37	40.52	54.47	397.95	1.39	4.1	18.33	12.33	19.48	57.9	72.81	490.3	153
Ghost	14	460	6.24	5.67	7.1	2.13	8.66	68.12	21.13	27.66	30.88	87.13	312.26	2.26	6.74	38.96	23.39	34.4	47.06	126.08	439.06	163
Peruggia	6	589	3.49	3.54	4.95	2.12	7.77	62.55	14.09	19.74	40.1	74.52	428.79	0.97	2.93	18.02	15.06	22.68	50.57	92.54	514.5	217
Samate	7	164	1.18	1.46	4.66	1.25	4.91	36.02	8.55	12.21	42.75	43.32	406.59	0.89	3,00	8.53	9.44	15.21	61.13	51.85	449.25	153
WackoPicko	36	2,099	17.43	14.25	25.74	9.42	32.88	263.35	66.84	90.29	35.09	320.77	379.92	0.45	1.26	1.84	67.29	91.56	36.06	322.61	379.44	301
Zipec	7	588	4.16	04.06	5.16	2.27	7.27	53.83	15.66	20.65	31.87	67.22	329.18	0.31	1.12	2.92	15.98	21.78	36.31	70.15	339.04	169
Total	**80**	**4,251**	**35.22**	**31.83**	**54.35**	**19.12**	**68.89**	**538.73**	**140.51**	**190.29**	**35.42**	**660.13**	**369.79**	**6.29**	**19.2**	**88.64**	**146.8**	**209.48**	**42.7**	**748.77**	**410.05**	**1,269**

execution, 94 PHP files (F) were ready for processing. However, only 80 were processed effectively by the tool, as the remaining 14 files contained object-oriented and advanced PHP instructions that the tool can not handle yet.

ITL Converter. In CoCoA's implementation, the ITL Converter module includes the Lexer and the ITL Translator, so we evaluate these components separately. The Lexer took approximately 32 ms to process all 7 webapps, which consisted of over 4,200 lines of code (LoC) spread over 80 files. As expected, the execution time tends to increase with the number of LoC and files. More LoC implies more tokens and data flows to process, and more files implies more system calls, which is also time-consuming. For the ITL Translator (Trans), some apps took approximately the same time to process as in the Lexer, while others took almost double the time. We believe this is due to the number of LoC and also the number of tokens per LoC in those applications, as this module needs to convert all LexTokens to ITL.

Data & Control Flows Extractor and Encryptor. We tested the Data & Control Flows Extractor in three scenarios: (i) with the Encryptor module disabled (DCF -Enc in the table); (ii) with *DET* and *RND* encryption of the index (DCF +Enc); and (iii) additionally with *ORE* encryption of line numbers and control flow (DCF +ORE). We separated the evaluation of ORE as it increases overhead by a large fraction.

As expected, the Data & Control Flows Extractor by itself adds little overhead to the solution. However, with *DET* and *RND* encryptions, this overhead grows by a large percentage, as many encryptions have to be done per index entry. Still, this overhead can be considered practical, as even in the worst case (WackoPicko webapp, with 36 files and 2,099 LoC) this module took only 32 ms to conclude. *ORE* encryptions, however, can increase overhead by an order of magnitude in some cases. This is due to the complexity of the encryption algorithm and also the number of encryptions (four) per index entry.

The column *Code Privacy Phase Total* in Table 5 shows the sum of all steps described so far, with additional encryption (*DET* and *RND*) and *ORE* overheads as a percentage, for the convenience of the reader.

Vulnerability Detector. This module was tested with an XSS analysis task and similarly to the Data & Control Flows Extractor: without encryption, with *RND* and *DET* encryptions, and additionally with *ORE*. Looking at Table 5, we can verify, once again, that *DET* and *RND* encryptions cause execution time to increase by some fraction and *ORE* by an order of magnitude in some cases. However, since this module is very fast, its overhead, even with *ORE*, can still be considered practical, with 38 ms in the worst case (Ghost) and 88 ms total with all webapps. Interestingly, we could also observe that this module's execution time is strictly related to the number of sensitive sinks (e.g., `XSS_SENS`) present in the application.

Total Execution Time. The total execution time of CoCoA was also measured without encryption, with *DET* and *RND*, and additionally with *ORE*, along with overhead percentages. Results show that the tool took a total of 146 ms to process all applications without encryption and 209 ms with *DET* and *RND*, i.e., an encryption overhead, on average, of 42.7%. With *ORE*, total overhead increased to 410.05%, totalling 748 ms to process all webapps and perform the XSS analysis.

Although these overheads may indicate some drawback of CoCoA, it should be noted that a large part of its execution time comes from the code privacy phase (see columns 10-14 of Table 5), which typically will only be executed once per app. Thus, of the encryption overheads, 35.42% and 369.79% are associated with this phase, respectively. The remaining overhead comes from the encrypted code analysis phase (7.28% and 40.26%, respectively), which is low and practical for the benefit gained in privacy.

Comparison with WAP. The last column of Table 5 shows the time that WAP took to process the same webapps. CoCoA revealed to be much more efficient than WAP;

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

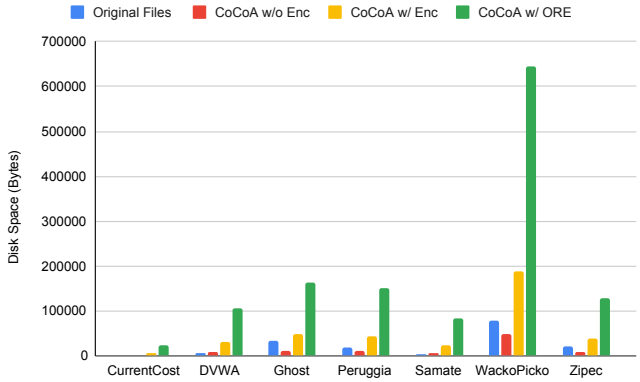

Fig. 5. Storage space used by the source code and the encrypted data structure, in bytes.

even with *ORE* encryptions. However, we believe the main reason for this discrepancy is that WAP not only detects vulnerabilities, but also predicts whether those vulnerabilities are real. Nonetheless, we can observe that WAP's execution time increases with LoCs in a similar fashion as our solution.

8.2.3 Memory and Storage Overhead

Lastly, we compared the storage space required by the source code of the applications with that of the index produced by CoCoA without encryption, with *DET* and *RND*, and with *ORE*. Figure 5 illustrates the results of these scenarios. CoCoA's index without encryption occupied less space than the original webapps; this is to be expected since CoCoA simplifies the code structure, discards irrelevant data, comments, and so on. Exceptions to this were the smallest applications with less than or equal to 7 KB, like Samate. Nonetheless, by adding *DET* and *RND* encryptions, the index size grew by some fraction due to the ciphertext expansion of these ciphers (e.g., 78 KB of source code to 49 KB of plaintext index and 188 KB of encrypted index in WackoPicko), and again by an order of magnitude with *ORE* (645 KB in WackoPicko).

9 CoCoA Extensibility

In this section, we give an overview on how CoCoA can be further extended to other analysis tasks and languages.

Analysis Tasks. Currently, CoCoA's prototype performs two analysis tasks – detection of SQLi and XSS vulnerabilities. It can be extended to other classes or even to a different type of analysis task. In both cases, the ITL-tokens that will represent the task and the code elements associated with that task should be added to the *Vuln. Knowledge* database. For example, to include the detection of Remote File Inclusion (RFI) vulnerabilities, the ITL-token `RFI_SENS` should be created, and the PHP code properties `include` (and others related to RFI) should be associated with this ITL token. To accommodate a new type of analysis task, e.g. type checking related to integer overflows, ITL-Tokens should be created to denote the integer data type (e.g., `INT_TYPE`), and arithmetic plus operations (+) should be considered throughout the data flow analysis to check if any exceeds the maximum value allowed for the data type.

Other Programming Languages. Extending CoCoA to other programming languages requires a set of additions. Nonetheless, the existing ITL-tokens can be reused, as well as the DCFG data structure and its encryption method. More concretely, the *ITL Tokens & Rules* and *Vuln. Knowledge* databases need to be constructed with the code properties and tasks of the new language. For example, to extend CoCoA to Java, we can reuse the `INPUT` ITL-token and map it to the `getParameter` entry point . Other tokens, such as operators, reserved words (`IF`, `WHILE`), and variable identifiers, could be reused without changes.

10 Related Work

Static Analysis. Static analysis aims to find bugs in code without executing it [43], [51], by checking the code of a program [18], [20], [44], [45]. Usually, static analysers build a model of the code through lexers to break the code into tokens [16], [45], parsers to generate code representation structures [24], [45] that the analyser will traverse to execute a set of rules about something it intends to find and a knowledge base that contains the data that characterise what it will look for in the code (e.g., vulnerabilities). Data flow analysis is the most used method to traverse these structures, and taint analysis is the most common technique used to discover bugs and extract the associated data flows [20], [33], [45]. Pixy [36], PHPCorrector [48], RIPS [16], and WAP [45] use this analysis to verify PHP code, efficiently detect sources and sinks, and analyse the data flow in the program through intra and inter-procedural techniques, resulting in accurate results. These tools must access the source code in cleartext, unlike CoCoA, and therefore do not provide code privacy.

Code Obfuscation and Encryption. Code obfuscation and encryption are used to make code unreadable [31]. The reasoning behind this is to protect the source code from a possible copyright violation (e.g., a company reusing the code of another company) and reverse engineering analysis by making it illegible [4], [11], [19], [37]. There are several ways to obfuscate code [31], [57], [70], [71], including changing variable names, modifying the order in which the code is written, and relying on jumps to retain the correct execution order [4]. Although this technique seems effective in ensuring code privacy, it is reversible, as it does not use encryption to obfuscate the code. In addition, this practice can make static analysis infeasible or less reliable since these tools are typically designed to parse the original source code without any form of obfuscation [60]. In contrast, code encryption has been used to ensure code privacy for cloud storage purposes [19] or to avoid reverse engineering analysis [6], [11], in which the entire or partial code (e.g., entry points and sensitive sinks) is encrypted and tested against tampering attacks. Code task analysis is allowed, but during the analysis process, the code is decrypted. CoCoA aims to protect the code with encryption and, simultaneously, enable static analysis over encrypted data using SSE, without revealing the code. During the code analysis of CoCoA, when decrypting the `value_ind` of the EII, what is found are the cryptographic keys of the next EII entry and the numeric data encrypted with ORE; so nothing is revealed about the code. Furthermore, these keys are the representative form of ITL tokens, and ITL tokens, in turn, are a form of obfuscation of the actual source code.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

Confidential Computing. There are many encryption schemes that allow computations on encrypted data. Both Fully Homomorphic Encryption (FHE) [28] and Secure Multi-Party Computation (MPC) [13] support additions and multiplications on encrypted data and hence can be used to execute any computation in the encrypted domain. However, these computations must be represented as arithmetic/boolean circuits, which can be highly inefficient in some cases. Moreover, FHE requires large CPU overhead and MPC large bandwidth overhead. Searchable Symmetric Encryption (SSE) [15], [25], [26] allows searching encrypted text databases efficiently, although revealing some information with each query. Oblivious RAM (ORAM) [63] eliminates this leakage, but at the cost of high bandwidth overhead. SSE has also been used to encrypt graphs and perform shortest distance queries [29], [47]. This is somehow similar to our approach; however, performing static analysis on a DCFG is different from performing a shortest-distance query on a graph, so an original approach had to be designed to solve our problem.

11 Conclusions

The paper presented the initial study of *Confidential Code Analysis*, a new research field that enables static code analysis tasks, such as vulnerability detection, in the encrypted domain while preserving code privacy. Also, CoCoA is presented, the first tool in this field to detect vulnerabilities in PHP web applications. The tool combines static analysis and SSE to obtain an encrypted data structure in the form of an inverted index that represents the program's data and control flow in a way that allows searches on it, i.e., the execution of code analysis tasks as a data flow analysis considering control flow. CoCoA was tested on synthetic and real PHP web applications, and the results showed that it achieved 80% and 93% precision repectively, and overall better results than two other static analysis tools. The analysis tasks over encrypted code take no longer than on clear data, and the encryption overhead employed by the tool is minimal. Also, the proposed data structure to store the encrypted code requires less disk space than the source files.

Acknowledgments

This work was supported by P2030 through project I2DT, ref. COMPETE2030-FEDER-00389100, an ITEA4 European project (ref. 22025), and by FCT through the APOSTLE project (ref. 2023.12254.PEX) and the LASIGE Research Unit (ref. UID/00408/2025).

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Longman Publishing Co., Inc., 2nd Edition, 2006.

[2] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing", *Communications of the ACM*, 2011, vol. 53, no. 4, pp. 50–58.

[3] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities", *Proceedings of the 2017 IEEE European Symposium on Security and Privacy*. IEEE, 2017.

[4] S. Banescu and A. Pretschner, "A tutorial on software obfuscation", Advances in Computers, 2018, vol. 108, pp. 283–353.

[5] D. Beazley, "PLY - Python Lex-Yacc", https://github.com/dabeaz/ply, 2025.

[6] J. Cappaert, N. Kisserli, D. Schellekens, and B. Preneel, "Self-encrypting code to protect against analysis and tampering", *Proceedings of the 1st Benelux Workshop on Information Systems Security*, 2006.

[7] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation", *Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium*. Internet Society, 2014.

[8] P. Chatterjee, A. Chatterjee, J. Campos, R. Abreu, and S. Roy, "Diagnosing software faults using multiverse analysis", *Proceedings of the 2021 International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

[9] Check Point Team, "Check Point Research Reports Highest Increase of Global Cyber Attacks seen in last two years – a 30% Increase in Q2 2024 Global Cyber Attacks", https://blog.checkpoint.com/research/check-point-research-reports-highest-increase-of-global-cyber-attacks-seen-in-last-two-years-a-30-increase-in-q2-2024-global-cyber-attacks/, 2024.

[10] B. Chess and G. McGraw, "Static analysis for security", *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[11] S. Cho, D. Shin, H. Jo, D. Choi, D. Won, and S. Kim, "Secure and efficient code encryption scheme based on indexed table", *ETRI Journal*, vol. 33, no. 1, pp. 60–70, 2011.

[12] CoCoA Development Team, "CoCoA tool and materials", https://github.com/iberiam/CoCoA, 2025.

[13] R. Cramer, I. B. Damgård *et al.*, "Secure multiparty computation", Cambridge University Press, 2015.

[14] CurrentCost development team, "DIY - Current Cost Home Energy Monitoring", https://github.com/r45635/CurrentCost2NRF-Public, 2015.

[15] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions", *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2006.

[16] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis", *Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium*. Internet Society, 2014.

[17] J. Daley, "Insecure software is eating the world: Promoting cybersecurity in an age of ubiquitous software embedded systems." *Stanford Technology Law Review*, vol. 19, no. 3, 2017.

[18] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution", *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[19] Y. Dong, A. Milanova, and J. Dolby, "Jcrypt: Towards computation over encrypted data", *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*, 2016.

[20] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: identifying vulnerable code for vulnerability assessment through program metrics", *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.

[21] DVWA development team, "Damn Vulnerable Web Application (DVWA)", https://github.com/digininja/DVWA, 2026.

[22] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis", *IEEE Software*, vol. 19, no. 1, pp. 42–51, 2002.

[23] FastORE development team, "FastORE", https://github.com/kevinlewi/fastore, 2016.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization", *ACM Trans. on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[25] B. Ferreira and H. Domingos, "Searching private data in a cloud encrypted domain", *Proceedings of the 10th Conference on Open Research Areas in Information Retrieval (OAIR)*, 2013.

[26] B. Ferreira, J. Leitão, and H. Domingos, "Muse: Multimodal searchable encryption for cloud applications", *Proceedings of the 37th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2018.

[27] B. Ferreira, B. Portela, T. Oliveira, G. Borges, H. Domingos, and J. Leitão, "Boolean searchable symmetric encryption with filters

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

on trusted hardware", *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 1307–1319, 2020.
[28] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.
[29] E. Ghosh, S. Kamara, and R. Tamassia, "Efficient graph encryption scheme for shortest path queries", *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.
[30] Ghost development team, "Ghost Framework", https://github.com/nk-o/ghost-framework, 2025.
[31] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, and V. Leppänen, "Diversification and obfuscation techniques for software security: A systematic literature review", *Information and Software Technology*, vol. 104, pp. 72–93, 2018.
[32] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo, "Verifying web applications using bounded model checking", *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004.
[33] ——, "Securing web application code by static analysis and runtime protection", *Proceedings of the 13th ACM International Conference on World Wide Web (WWW)*, 2004.
[34] J. Levine, T. Mason, and D. Brown, Lex & Yacc, O'Reilly & Associates, Inc., 2nd Edition, 1992, vol. 2.
[35] T. Ishio, "Dependency-based extraction of conditional statements for understanding business rules", *IEICE Transactions on Information and Systems*, vol. 99, no. 4, pp.1117–1126, 2016.
[36] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities", *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, ACM, 2006.
[37] S. Kang, S. Lee, Y. Kim, S.-K. Mok, and E.-S. Cho, "Obfus: An obfuscation tool for software copyright and vulnerability protection", *Proceedings of the 11th ACM Conference on Data and Application Security and Privacy*, 2021.
[38] J. Katz and Y. Lindell, "Introduction to modern cryptography: principles and protocols", Chapman and hall/CRC, 2007.
[39] C. Y. Laporte and A. April, Software quality assurance, John Wiley & Sons, 2018.
[40] M.-C. Lee, "Software quality factors and software quality metrics to enhance software quality assurance", *British Journal of Applied Science & Technology*, vol. 4, no. 21, pp. 3069–3095, 2014.
[41] K. Lewi and D. J. Wu, "Order-revealing encryption: New constructions, applications, and lower bounds", *Proceedings of the 2016 ACM Conference on Computer and Communications (CCS)*, 2016.
[42] C. D. Manning, "Introduction to information retrieval", Syngress Publishing, 2008.
[43] D. Marcilio, R. Bonifácio, E. Monteiro, E. D. Canedo, W. P. Luz, and G. Pinto, "Are static analysis violations really fixed?: a closer look at realistic usage of sonarqube", *Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2019.
[44] I. Medeiros, N. Neves, and M. Correia, "DEKANT: a static analysis tool that learns to detect web application vulnerabilities", *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016.
[45] ——, "Detecting and removing web application vulnerabilities with static analysis and data mining", *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016.
[46] ——, "Statically detecting vulnerabilities by processing programming languages as natural languages", *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 1033–1056, 2022.
[47] X. Meng, S. Kamara, K. Nissim, and G. Kollios, "Grecs: Graph encryption for approximate shortest distance queries", *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.
[48] R. Morgado, I. Medeiros, and N. Neves, "Towards web application security by automated code correction", *Proceedings of the 2020 International Conference on Evaluation of Novel Approaches to Software Engineering*, 2020.
[49] M. Naeem, A. Tahoon, O. Ragi, and R. El-Adawi, "The secret to Calibre software quality – AnaCov, our in-house code coverage analysis tool", https://blogs.sw.siemens.com/calibre/2024/03/07/the-secret-to-calibre-software-quality-anacov-our-in-house-code-coverage-analysis-tool/, 2024.
[50] L. H. Newman, "The tricky aftermath of source code leaks", WIRED, https://www.wired.com/story/source-code-leak-dangers/, 2022.
[51] F. Nielson, H. R. Nielson, and C. Hankin, Principles of program analysis, Springer, 1999.
[52] NIST, "SAMATE - Software Assurance Metrics and Tool Evaluation", https://samate.nist.gov/, 2024.
[53] NIST, "Minimum security requirements for federal information and information systems", FIPS PUBS 200, U.S. Department of Commerce, 2006.
[54] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization", *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2017.
[55] F. Pecorelli, S. Lujan, V. Lenarduzzi, F. Palomba, and A. De Lucia, "On the adequacy of static analysis warnings with respect to code smell prediction", *Empirical Software Engineering*, vol. 27, no. 64, pp. 1–44, 2022.
[56] Peruggia development team, "Peruggia", https://github.com/Sakartu/peruggia, 2015.
[57] PHP Obfuscator, "Online obfuscator for PHP to make it difficult to understand.", https://php-minify.com/php-obfuscator/, 2026.
[58] QA Mentor, "White-box Testing", https://www.qamentor.com/testing-coverage/functional-testing/white-box-testing/, 2025.
[59] R. N. Rajapakse, M. Zahedi, and M. A. Babar, "An empirical analysis of practitioners' perspectives on security tool integration into devops", *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021.
[60] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?", *Acm Computing Surveys*, vol. 49, no. 1, pp. 1–37, 2016.
[61] ScienceSoft, "Application Security Assessment - Detecting and Remediating App Vulnerabilities", https://www.scnsoft.com/security/assessment/applications, 2026.
[62] SecurityScorecard, "CVEdetails.com - Browse CVE Vulnerabilities By Date", https://www.cvedetails.com/browse-by-date.php, 2026.
[63] E. Stefanov, M. v. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol", Journal of the ACM (JACM), vol. 65, no. 4, pp. 1–26, 2018.
[64] A. van der Stock, B. Glas, N. Smithline, T. Janca, and T. Gigler, "OWASP Top 10:2025", https://owasp.org/Top10/, 2025.
[65] D. X. Song, D. A. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data", *Proceeding of the 2000 IEEE Symposium on Security and Privacy (S&P)*, 2000.
[66] Veracode, "State of Software Security 2023: Annual Report on the State of Application Security", https://info.veracode.com/rs/790-ZKW-291/images/Veracode_State_of_Software_Security_2023.pdf, 2023.
[67] ——, "Veracode: Application Risk Management Engineered for the AI-Coding Era", https://www.veracode.com/, 2026.
[68] W3Techs, "Usage statistics of server-side programming languages for websites", https://w3techs.com/technologies/overview/programming_language, 2026.
[69] Wackopicko development team, "WackoPicko Vulnerable Website", https://github.com/adamdoupe/WackoPicko, 2021.
[70] M. Wu, Y. Zhang, and X. Mi, "Binary protection using dynamic fine-grained code hiding and obfuscation", *Proceedings of the 4th International Conference on Information and Network Security*, 2016.
[71] YAK Pro development team, "YAK Pro - Php Obfuscator", https://www.php-obfuscator.com/?lang=english, 2025.
[72] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs", *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P)*, 2014.
[73] YAML development team, "YAML ain't markup language", https://yaml.org/, 2021.
[74] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines", *Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2017.
[75] Zipec development team, "ZiPEC - iPhone Event Console for Zenoss", https://sourceforge.net/projects/zipec/, 2015.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2026.3688318

Jorge Martins is a Software Engineer at Critical TechWorks - BMW Group - Portugal. He holds an MSc in Computer Engineering from the Faculty of Sciences of the University of Lisbon (FCUL). He is a young and passionate software engineer with an interest in Software Development Security.

David Dantas is a Software Engineer at ComplyAdvantage, Portugal. He holds an MSc in Computer Engineering from the Faculty of Sciences of the University of Lisbon (FCUL), where he also completed his BSc, and is a researcher of the LASIGE research unit. His thesis was realted to cybersecurity, focusing on privacy-preserving systems, as part of the SMaRtChain project. His main interests include cybersecurity and distributed systems.

Rafael Ramires is a PhD student of Informatics at the Faculty of Sciences of the University of Lisbon (FCUL) and the University of Luxembourg, doing research in the software security area. He holds an MSc in Computer Engineering from FCUL. He has been involved in activities for the detection of vulnerabilities in web applications in the SEAL, XIVT and I2DT projects. His research interests include software security, artificial intelligence and multi-agent systems.

Bernardo Ferreira is an Assistant Professor with the Faculty of Sciences, University of Lisbon, Portugal, and an integrated researcher with the LASIGE research unit. He received the BSc, MSc, and PhD degrees in computer science from the Faculty of Sciences and Technology, Nova University of Lisbon, Portugal, in 2008, 2010, and 2016, respectively. His research interests include distributed systems security and privacy, with special focus on blockchains and secure distributed computation (Homepage: https://www.di.fc.ul.pt/~blferreira/).

Ibéria Medeiros is an Associate Professor at the Faculty of Sciences of the University of Lisbon (FCUL), and an integrated researcher of the LASIGE research unit. Her research interests are software security, vulnerability and attack detection, code privacy and correction, and artificial intelligence applied to cybersecurity. She is the author of tools for software security and cybersecurity, and has been involved in international and national projects. More information about her at https://www.di.fc.ul.pt/~imedeiros/.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 06,2026 at 13:20:40 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.