DAI: A Dependencies Analyzer and Installer For Solidity Smart Contracts

Giacomo Ibba[1], Giuseppe Destefanis[2], Rumyana Neykova[2], Marco Ortu[1],
Sabrina Aufiero[3], Silvia Bartolucci[3]
[1]*University of Cagliari, Italy*
[2]*Brunel University London, UK*
[3]*University College London, UK*
{*giacomo.ibba,marco.ortu,roberto.tonelli*}@*unica.it*
{*sabrina.aufiero.22,s.bartolucci*}@*ucl.ac.uk*
{*rumyana.neykova,giuseppe.destefanis*}@*brunel.ac.uk*

***Abstract*—The growing importance of Decentralized Applications (dApps) in areas such as the Internet of Things (IoT), Cybersecurity, and Finance is playing a crucial role in advancing software maintenance, security, and data sharing. Understanding the complex architecture and components of dApps is essential to harness their full benefits. This often involves the challenging task of identifying and retrieving key components during the dApp compilation process, particularly when dealing with multiple external dependencies. A case in point is the variety of versions in the OpenZeppelin libraries, where finding compatible elements can be a laborious process. In response to this challenge, we introduce DAI (Dependency Analyser and Installer), a novel tool that automates the identification of compatible external dependency versions for specific smart contracts. This tool significantly simplifies the compilation process for dApps that incorporate external modules, making it more efficient for developers and researchers. We evaluated DAI on 57 real-world dApps, achieving success in determining the right dependency match for 50 cases. However, the inability to compile the remaining 7 dApps due to missing files and artifacts highlights the ongoing complexities in dApp development.**

***Keywords*—Blockchain, Smart Contracts, Decentralized Applications, Software Engineering**

I. Introduction

The increasing relevance of Decentralized Applications (dApps) is revolutionizing sectors where network control traditionally rests with a single node. These applications, free from central management, offer improved security and transparency, significantly enhancing the scalability of IoT and financial applications. Understanding the architectural components of dApps is vital for effective software maintenance and development. Examining the architecture of a dApp involves a thorough analysis of source code artifacts, including external dependencies. For instance, Solidity-based dApps often rely on OpenZeppelin libraries, which are essential for increasing security, scalability, and preventing vulnerabilities like overflows, underflows, and reentrancy attacks. One of the critical steps in dApp development is extracting key components, sometimes requiring the compilation of the dApp. Compiling dApps without external dependencies is relatively simple, but those with external dependencies, such as OpenZeppelin libraries, present unique challenges.

To illustrate the above challenge, consider Figure 1, which shows the web of dependencies and versions that characterizes typical dApps. Each library poses different constraints and brings different paths, contracts, and functions, which can lead to compatibility issues with the dApp's chosen compiler version. Identifying the correct dependency version that aligns with the dApp is a complex task. The complexity of identifying correct dependency versions is further amplified when dealing with large datasets of smart contract codebases, each characterized by a unique set of requirements.

To tackle this problem, we developed DAI (Dependency Analyzer and Installer). This tool automatically detects and installs the right versions of external dependencies for dApps. By enabling the compilation and production of binaries from datasets with complex dependencies, DAI expands the applicability of these datasets for the evaluation of both static and dynamic analysis tools. It allows for the effective use of purely codebase-driven datasets.

We tested DAI on a varied set of dApps, each importing different versions of external dependencies. The tool effectively identified the compatible versions in most cases, with the exceptions of applications where missing files hindered compilation.

II. Related Work

Recent literature has explored various aspects of the development of blockchain-based decentralized applications (dApps). Here we mention only a few studies to give a general understanding of the plethora of applications of smart contract analysis. Udokwu et al. [4] conducted an evaluation of existing design methodologies, emphasizing the importance of robust frameworks in the development of DApps. This is complemented by the work of Duan et al. [2], who have focused on testing techniques and tools specifically targeting DApps development, highlighting the unique challenges they present. Bartl [1] provides an analysis of utilization trends in dApps, offering valuable statistical data on the adoption

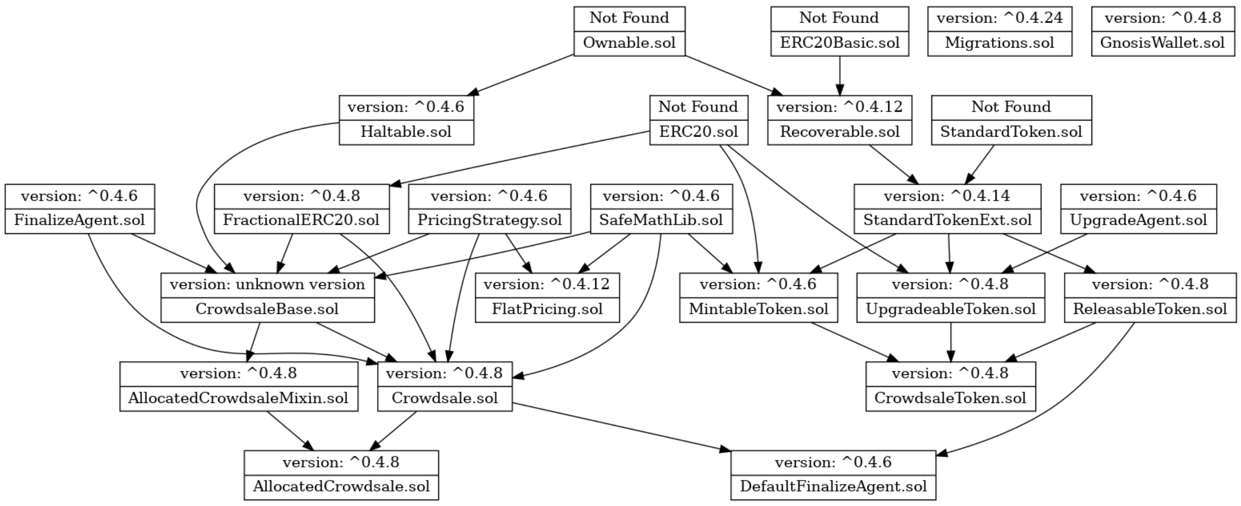

Fig. 1: Dependency relations of a DAppsample (Figure 4 [5])

and impact of these applications. These studies contribute to our understanding of the design, testing, and usage patterns of dApps, but also highlight the importance of large-scale analysis of related codebases.

Other related works explore architectural components and dApps' structure. For example, MindTheDApp [3] is a toolchain designed specifically for the understanding of Ethereum-based applications structure and operational logic, with a distinct focus on a complex network-driven approach. All of the above tools for smart contract analysis rely on existing datasets of smart contract codebases.

In the context of Ethereum smart contract development, environments like Truffle [1], Hardhat [2] and Ethereum Package manager [3], automate aspects of building, testing, and deploying dApps, but their capabilities are insufficient for analyzing large datasets where dependencies vary greatly. Truffle provide a suite of tools to simplify and streamline the process of developing, testing, and deploying smart contracts on the Ethereum blockchain. Hardhat as well provides an environment for dApps development and customization, but unlike Truffle has more advanced testing capabilities, which can be helpful for writing more comprehensive and robust tests. However, as for Truffle or the Ethereum Package Manager (ethPM), developers may not release all the artifacts of the application, like the configuration files, which are crucial to gather information about dependencies versions. Neither of these tools can be used to compile smart contracts from diverse source-code-only datasets.

The motivation behind the development of our tool is to enable researchers to perform large-scale smart contract analysis, which requires large datasets of programs relying on complex dependencies. By simplifying the process of compilation, the analysis can be performed on the contract codebase, without requiring the contract binaries. To the best of our knowledge, our Dependencies Analyzer and Installer (DAI) is the first tool to analyze and suggest the compatible dependency version (which is the one granting the compilation of the whole application) for a specific decentralized application.

III. Research Methodology

A critical initial step was to secure real-world dApps for tool testing. We referred to the work of [5] which introduced DappScan, a collection of buggy decentralized applications. An examination of this dataset revealed a sample of 550 dApps incorporating external dependencies, from OpenZeppelin, PancakeSwap, Uniswap, Opensea, and other libraries. For this phase of our study, we focused solely on dApps using OpenZeppelin contracts as external dependencies, with plans to extend DAI's capabilities to additional dependencies in future work.

The architecture of the tool is depicted in Figure 2. It features (1) a module for extracting the pragma version of the compiler utilizing the ANTLR parser generator; (2) a module designed for selecting and altering the compiler version, and (3) a dependency linker and matcher that identifies the (openzeppelin) dependencies and adjusts the relative path of the imports in the contracts.

The process of scanning and analyzing dependencies is complex due to the variation in files and functionalities across different versions. Focusing on OpenZeppelin contracts, it was necessary to gather all available versions of the *openzeppelin/-contracts* dependency. We compiled a comprehensive JSON file listing every release of this particular dependency. For each major version (e.g., 0.5), we created a detailed JSON file that provides, for each specific release (like 0.5.0, 0.5.1), a list of paths and accessible files, along with the exact pragma version used in each file.

Once the scanning process is completed, DAI outputs a list of OpenZeppelin contract versions that are compatible with the tested dApp. This approach ensures a thorough and

[1]https://trufflesuite.com/
[2]https://hardhat.org/
[3]https://www.ethpm.com/

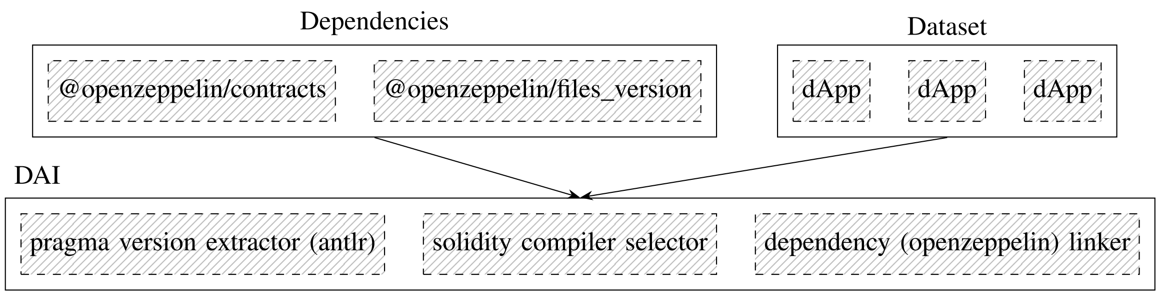
Dependencies
@openzeppelin/contracts
@openzeppelin/files_version
Dataset
dApp
dApp
dApp
DAI
pragma version extractor (antlr)
solidity compiler selector
dependency (openzeppelin) linker

Fig. 2: DAI Architecture Toolchain

accurate assessment of compatibility between the dApps and their dependencies.

A. Dataset

Our study utilized the DAppScan dataset, a comprehensive repository encompassing a diverse range of dApps with known vulnerabilities, both deployed and undeployed. Within this dataset, each dApp entry includes the source code, comprised of smart contracts, information on any identified vulnerabilities (if applicable), as well as professional audit reports provided by auditing firms

We selectively filtered the dApps that incorporated the *@openzeppelin/contracts* dependency. This process yielded a subset of 57 dApps, spanning various Solidity pragma versions. The version of Solidity used in the source files is a key determinant in establishing which dependency version is compatible with the decentralized application.

There are 79 reported versions of *@openzeppelin/contracts* available through *npm*, including beta versions. Through npm, it's feasible to install compatible dependency versions for dApps using Solidity pragma versions ranging from 0.5 to 0.8. However, applications operating with the 0.4 root version often necessitate obtaining the dependency directly from GitHub or creating custom contracts based on OpenZeppelin guidelines. Since retrieving or constructing these ad-hoc dependencies can be arduous, our current implementation of DAI does not support the 0.4 version.

B. Solidity And Dependency Versions

DAI currently accommodates Solidity versions from 0.5 to 0.8. It is important to note that compatibility issues can arise between different versions. For example, *@openzeppelin/contracts* designed for the 0.5 Solidity pragma version are incompatible with higher major versions, such as 0.6. Similarly, newer Solidity versions, like 0.8, are not backward compatible with earlier versions like 0.7.

Furthermore, there can be compatibility discrepancies within the same root version. For instance, OpenZeppelin contracts compatible with versions beyond 0.8.20 may not work with dApps designed for general versions above 0.8.0. These incompatibilities often stem from various updates in each release, including semantic and syntactic alterations, explicit requirements, and the phasing out of certain features.

C. Dependency Identification Process

Initially, DAI examines each smart contract within a dApp, focusing on those importing OpenZeppelin contracts. It extracts the Solidity pragma from these files, while Solidity files not importing OpenZeppelin dependencies are excluded from the analysis.

In some cases, a dApp might use modules with varying Solidity pragma versions for specific design and compatibility purposes. These modules might require different versions of the same dependency for successful compilation. DAI evaluates each Solidity file independently and identifies compatible dependency versions. However, the task of aligning each module with the appropriate version rests with the user.

Once DAI determines the version of a dApp, it searches for all related releases of *@openzeppelin/contracts* and assesses their compatibility. The following cases illustrate how DAI handles different pragma specifications:

- pragma solidity ˆ0.5.2;: A file with this line will compile with versions later than 0.5.2 but not with versions starting from 0.6.0. DAI will consider releases greater than 0.5.2 and less than 0.6.0.
- pragma solidity ∼0.7.0;: This indicates compatibility with versions of the Solidity compiler that align with 0.7.0, including minor updates and bug fixes, but excluding major changes.
- pragma solidity =0.8.2;: A file with this directive is intended to compile only with version 0.8.2.
- pragma solidity >0.7.0 <0.9.0;: This range specifies that the contract is compatible with compiler versions between 0.7.0 and 0.9.0. DAI must first identify the appropriate Solidity version for the contract before determining the compatible dependency version.

DAI's final step is to analyze the extracted possibilities, focusing on the compatibility between dependency versions and dApp imports. This involves a two-step analysis:

- Path Comparison: DAI compares the source file paths imported by the dApp with those in the dependency. It examines if the dependency's path matches the dApp's imported path. A mismatch indicates incompatibility between the current dependency version and the dApp's expectation.
- Pragma Version Check: DAI then verifies if the pragma versions in the dependency's source files align with

those in the dApp. Even if two dependency versions share the same root pragma and file paths, differing specific Solidity versions might lead to incompatibility. DAI evaluates these pragma versions against each other to confirm compatibility.

IV. Results and Discussion

Our testing of DAI focused on its ability to select compatible versions of *@openzeppelin/contracts* for various dApps. The primary method to verify compatibility was by compiling the dApps; successful compilation without errors indicated that DAI correctly identified compatible modules and dependencies.

We applied DAI to a subset of 57 dApps that import *@openzeppelin/contracts*. In 50 of these cases, DAI accurately detected the dependency versions that aligned with each dApp's Solidity version. The remaining 7 dApps could not be compiled due to missing files in their smart contracts, leading us to categorize these instances as unsuccessful analyses. This limitation stems from our reliance on an external dataset, which may not always include all necessary modules and artifacts.

Significantly, DAI proved effective even in complex scenarios, such as those involving compiler range specifications (e.g., >0.7.0 <0.9.0). All four dApps using these ranges were successfully scanned by DAI. Additionally, DAI handled dApps with modules importing different dependency versions. We validated DAI's output in these cases by manually linking the suggested *@openzepellin/contracts* versions to the respective modules before compilation.

V. Threats To Validity

In evaluating DAI, we must consider several factors that could affect its validity. A primary concern is the scope of our dataset. The 57 applications we used represent a specific segment of dApps and may not fully capture the broader complexity and range of artifacts present in the wider dApp ecosystem.

Another key aspect is the nature of our test cases. Currently, DAI's testing is confined to real-world scenarios. While valuable, this approach might overlook complex, hypothetical cases that could challenge and refine the tool further. Creating and testing these intricate, custom scenarios is a challenging yet necessary step for a comprehensive assessment.

The scope of *@openzeppelin/contracts* versions supported by DAI also poses a limitation. Presently, DAI does not accommodate dependencies that are compatible with the 0.4 Solidity version, which, despite not being installable via *npm*, remains relevant in certain contexts.

Additionally, DAI's current focus is limited to *@openzeppelin/contracts*, excluding other significant dependencies like *@openzeppelin/contracts-upgradeable* and *@openzeppelin/contracts-ethereum-package*. Moreover, popular dependencies such as *@pancakeswap*, *@uniswap* and *opensea* are not yet supported, which restricts the potential applications of our tool.

While DAI shows promise, these limitations highlight the need for ongoing development to broaden its applicability and enhance its robustness in diverse scenarios.

VI. Future Work and Conclusions

Our future efforts will focus on addressing the identified threats to DAI's validity and enhancing its applicability. A key goal is to expand our dataset by acquiring more diverse dApps. This expansion, coupled with the creation of custom test cases featuring complex imports, will enable a more thorough assessment of DAI's effectiveness.

A primary objective is to ensure DAI's compatibility with all versions of *@openzeppelin/contracts*, including those for Solidity 0.4. Keeping DAI updated with the latest versions of these contracts will also be a priority.

We plan to broaden DAI's scope by incorporating additional external dependencies commonly used in dApps. Integrating these dependencies will significantly enhance DAI's functionality, allowing for a more comprehensive analysis of a wider range of applications. Additionally, we aim to make DAI compatible with deprecated packages, recognizing that many deployed dApps may still depend on these older versions.

Another ambitious goal is to enable DAI to work with dependencies sourced from GitHub. This task presents unique challenges due to the difficulty in tracking dependencies available through such external platforms.

Our work aims to turn DAI into an essential tool in the toolbox of smart contract developers and researchers, helping them navigate the diverse and evolving landscape of blockchain technology. Our preliminary analysis has showcased that DAI's can compile dApps with a wide array of external dependencies and varying compiler versions. This not only highlights DAI's potential in assisting developers and researchers in pinpointing compatible dependency versions for their dApps but also its capability to handle and adapt to a richly varied set of smart contracts.

Acknowledgement

S.B., G.D., R.N. and M.O. acknowledge support from the Ethereum foundation grant FY23-1048.

References

[1] Mathias Bärtl. A statistical examination of utilization trends in decentralized applications. *Frontiers in Blockchain*, 6:01–12, 2023.
[2] Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. Towards automated safety vetting of smart contracts in decentralized applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 921–935, 2022.
[3] Giacomo Ibba, Sabrina Aufiero, Silvia Bartolucci, Rumyana Neykova, Marco Ortu, Roberto Tonelli, and Giuseppe Destefanis. Mindthedapp: A toolchain for complex network-driven structural analysis of ethereum-based decentralised applications. *IEEE Access*, pages 1–1, 2024.
[4] Chibuzor Udokwu, Henry Anyanka, and Alex Norta. Evaluation of approaches for designing and developing decentralized applications on blockchain. In *Proceedings of the 4th International Conference on Algorithms, Computing and Systems*, pages 55–62, 2020.
[5] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. Dappscan: Building large-scale datasets for smart contract weaknesses in dapp projects. *arXiv preprint arXiv:2305.08456*, 2023.