Master's Thesis: HATS

A Haskell Auditing Tool for Security

Author:	Supervisors:
T.J. Blok	Dr. M. Vasenna
6825990	Dr. G.K. Keller

July 9, 2024

Abstract

We present HATS, an auditing tool Haskell developers can use to inspect, query, and detect changes in functional dependencies, including transitive dependencies. We present this tool for Haskell, implemented in Haskell. Using a Core plugin, it analyses the program's call graph to find dependencies between functions. We demonstrate, using an example, how HATS can help detect attacks such as supply chain attacks.

Contents

1 Introduction

Software developers often re-use code written by others. Doing this also means using the dependencies of that code. However, it is not always clear to the user what transitive dependencies are used. Malicious actors exploit this fact with *supply chain attacks*[1], [2]. Here, vulnerable code (e.g. a backdoor) is injected in some transitive dependency and therefore in the final product as well. Another way vulnerable code can be injected is through attacks called *typosquatting*[3]. In such attacks, a malicious package, with a name suspiciously similar to a commonly used package (e.g. *crossenv* instead of *cross-env*), is uploaded to a package manager. The attacker then hopes developers accidentally misspell the desired package in a config file and thus use their package instead. To combat these attacks, developers must audit the packages their program depends on. However, the amount of transitive dependencies quickly grows too big to audit manually. Tools are needed to help security-conscious developers catch these suspicious code injections before they can cause harm.

Haskell has a strong type system, which is why much research has been done making security tools using this type system (discussed in section 3.2.1). However, these tools only focus on currently known vulnerabilities or are not backwards compatible with arbitrary existing code. Therefore, none of these tools can help developers find previously unknown cases of injected code in arbitrary code, including its transitive dependencies.

This paper presents HATS, a Haskell Auditing Tool. It is a static analysis tool, which helps developers audit their code and its dependencies. HATS uses the program's call graph to find injected code and the path the program takes to it. A call graph shows the program's flow, where the nodes are functions and the edges are references between them. It then filters this call graph before finally visualising the graph. It can filter out functions not defined in a specific module or package, and it can filter out pure functions (functions with no side effect). It can also track the changes in dependencies, allowing the developer to audit only the appropriate functions. Because HATS uses the call graph, it gives us finer precision when compared with detecting changes at the level of packages or modules. The main contributions of this paper include:

- We present HATS, an auditing tool Haskell developers can use to inspect, query, and detect changes in functional dependencies, including transitive dependencies.
- We demonstrate, using an example, how HATS can help detect attacks such as supply chain attacks.

2 Walkthrough/Overview

2.1 Example

Consider the following example. A trusted developer, Alice, wants to write an application where users can sign up and make accounts. To write this app, she must manage the user's passwords. She determined that the password manager package she found online would be ideal for this task. This password manager also uses a different package to check whether the chosen password is too common. Figure 1 shows how these packages depend on each other.

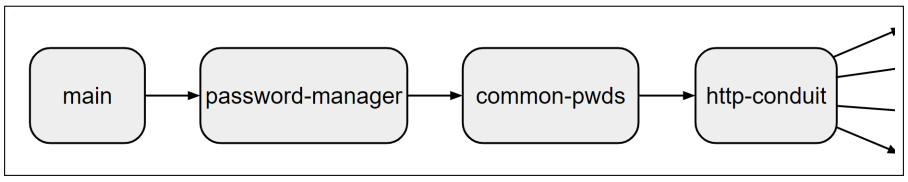


Figure 1: The package structure of the example. All shown packages depend on the base package, but it is left out for simplicity.

In this simple example we will mainly focus on packages *main*, *password-manager*, and *common-pwds*. Each of these packages has exactly one module, all shown below.

```haskell
module Main where
import PasswordManager (savePasswordToFile)

main = do
    password <- getLine
    savePasswordToFile password

    -- the rest of the app
    ...
```

```haskell
module PasswordManager where
import CommonPasswords (commonPasswords)

savePasswordToFile :: String -> IO ()
savePasswordToFile password = do
    common <- commonPasswords
    if (password `elem` common)
        then putStrLn "weak password!"
        else writeFile "password.txt" password
```

```haskell
module CommonPasswords where

import Network.HTTP.Conduit (simpleHttp)
import Data.ByteString.Lazy.Char8 (unpack)

-- | accesses external database for common passwords
commonPasswords :: IO [String]
commonPasswords = do
  bs <- simpleHttp "http://pwds.org/dict_en.txt"
  return $ lines $ unpack bs
```

2.2 Call Graph Visualisation

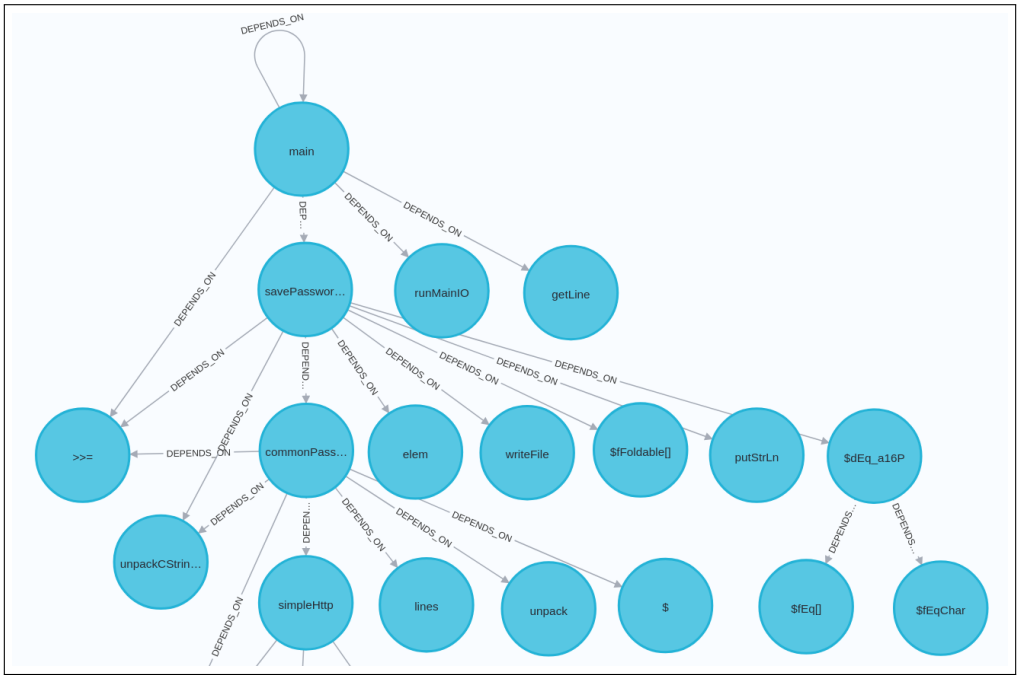
To audit these packages, let's first look at the call graph. Visualising the call graph can help grasp how the program flows. Our example is only a few modules big, but real-world packages often depend on dozens of other packages, each containing dozens of modules.
The call graph will include all functions the program might call (see figure 2). The arrows from *main* to *main* in figure 2 and 3 are artefacts from Core (The language used under the hood by GHC (the Glasgow Haskell Compiler). The user-defined *main* function gets called by a different function also called *main*. This true *main* function wraps the user-defined *main* with *runMainIO*, giving it exception handlers. But, because they share the same name, module, and package, they are merged in the graph.

Not all information inside this call graph may be important to the user. HATS can be set only to detect code interacting with the rest of the system or the internet. In Haskell, such actions are called IO (Input/Output) actions. IO actions include actions such as accessing the type system, or writing and reading from the terminal. Only functions with the IO type or FFI (Foreign Function Interface) bindings can run IO actions. FFI bindings allow a developer to run arbitrary C code, which doesn't have any concept of pure functions.

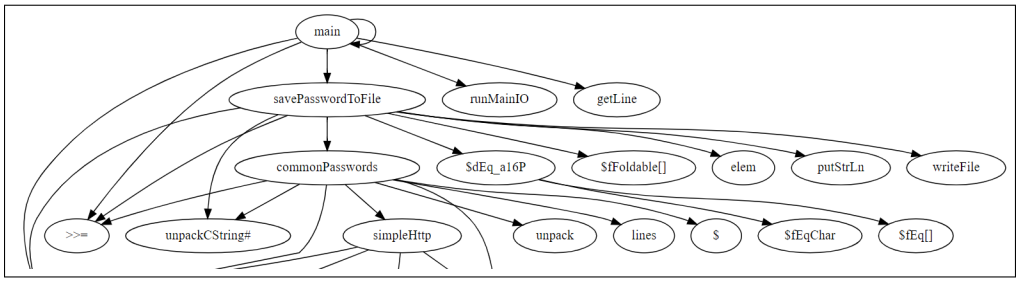
Whenever the IO behaviour of a program changes, it could mean a vulnerability is introduced, such as a data leak. However, certain vulnerabilities can still be introduced without needing IO actions. Examples of this are timing attacks, where the attacker uses the execution time of a program to leak data. In our example, Alice is only concerned with changes in IO behaviour. Therefore, she filters the graph, such that only paths leading to IO functions or FFI bindings are left (see figure 3).

2.3 Tracking Changes

When Alice is done auditing and chooses to use this package, she can use the *track* option from HATS. This outputs all the nodes that the declaration we track depends on into a text file. A declaration, e.g. the function *savePasswordsToFile*, is defined by its package, module, and occurrence name. This allows Alice to keep an eye on the dependencies of specific declarations. She can then use this to



(a) Neo4j interactive environment.

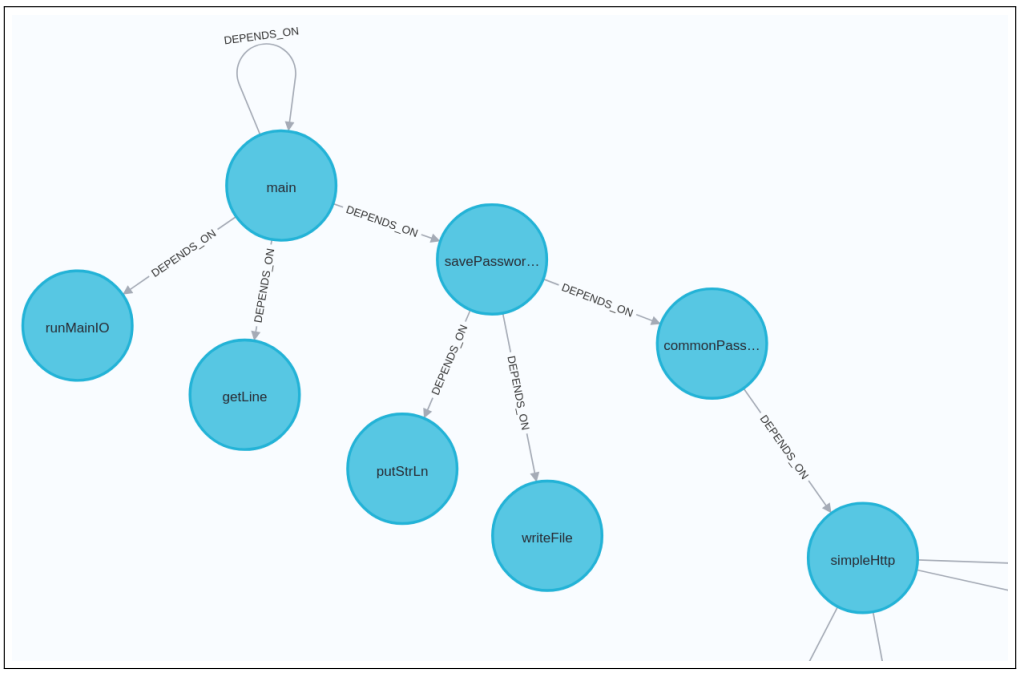


(b) Graph Viz.

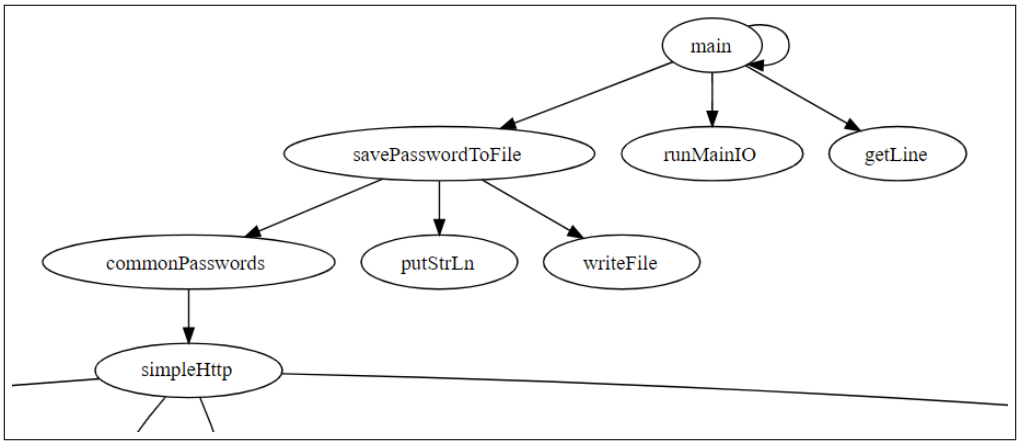
Figure 2: A comparison, between Neo4j and Graph Viz, of the call graph *before* being filtered for IO functions. Zoomed in on the part of the graph relevant to the example

easily detect changes in the dependencies by analysing the difference after each build. When a package changes, this could potentially mean a vulnerability has been introduced. However, programs rarely utilise the full capability of a package, let alone the package it transitively depends on. When the package has an update, she will know whether it affects her usage of that package or not. This means she only needs to audit the declarations that are relevant to her.

Let's see what this would look like in the example. Imagine a malicious actor, named Bob, managed to insert a few lines of code into the new version of the *common-pwds* package. The edited module



(a) Neo4j interactive environment.



(b) Graph Viz

Figure 3: A comparison, between Neo4j and Graph Viz, of the call graph *after* being filtered for IO functions. Zoomed in on the part of the graph relevant to the example

now looks like this:

```haskell
module CommonPasswords where

import Network.HTTP.Conduit (simpleHttp)
import Data.ByteString.Lazy.Char8 (unpack)

-- | accesses external database for common passwords
commonPasswords :: IO [String]
commonPasswords = do

  -- two added malicious lines
  pwd <- readFile "password.txt"
  simpleHttp ("http://bob.evil/pwd=" ++ pwd)

  bs <- simpleHttp "http://pwds.org/dict_en.txt"
  return $ lines $ unpack bs

```

In this example, the password manager has configured its dependencies to always use the latest version. This means when Alice tries to rebuild her package, the package manager will use the latest version of the *common-pwds* package. When this happens, these malicious lines of code are also automatically added to Alice's build. However, because Alice used HATS with her build, both previously and this time, keeping an eye on the declaration *savePasswordToFile*. She checks the difference between her tracking files. This can be done using any simple diff tool and comparing the two text files.

```
$ diff tracking-results-old.txt tracking-results.txt
_6c6_
<  http-conduit:Network.HTTP.Conduit:simpleHttp is referenced by
↪  common-pwds:CommonPasswords:commonPasswords 1 time(s).
---
>  http-conduit:Network.HTTP.Conduit:simpleHttp is referenced by
↪  common-pwds:CommonPasswords:commonPasswords 2 time(s).
_3509a3510_
>  base:System.IO:readFile is referenced by common-pwds:CommonPasswords:commonPasswords 1 time(s).
```

Alice finds that *simpleHttp* has been referenced one more time than last time and a new dependency has been added (indicated by the *a* in the result of the diff tool). This raises alarm bells for her because she did not expect *commonPasswords* to need to use *readFile*. The change she found doesn't have to be malicious, but at least she knows exactly where to look. The diff tool output already tells Alice which declaration references *readFile*, but she would like to know what path the program takes to this new dependency. She can do this by importing the call graph into Neo4j, which then allows Alice to query the graph using the following Cypher query. Note that unit is a synonym for package.

```
MATCH path=(decl:Declaration {name:"savePasswordToFile",
    module:"PasswordManager", unit:"password-manager"})-[*0..]->
    (target:Declaration {name:"readFile", module:"System.IO", unit:"base"})
RETURN path
```

This query will find the exact paths the program takes from *savePasswordToFile* this newly added dependency. Figure 4 shows the result of this query in Neo4j.

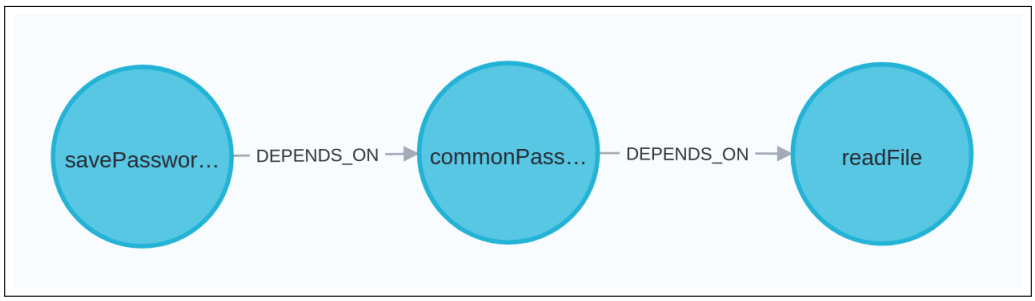


Figure 4: The path from the declaration Alice was tracking, to the newly added dependency.

3 Related Work

This section will discuss a few security tools in Haskell and other languages. Security tools either work at run-time or compile-time. Table 1 compares all the tools discussed in this section.

3.1 Vulnerability Scanners

Vulnerability scanners, such as Snyk[4] and Cabal Audit[5], scan source code and its dependencies to find vulnerable functions. For this, they rely on external databases to have knowledge of which functions contain a vulnerability. HATS aims to help find these vulnerabilities, without needing a vulnerability database.

Access control tools	Mir	Shill	LIO	HLIO	MAC	Safe Haskell
Approach	Dynamic	Dynamic	Dynamic	Hybrid	Static	Static

(a) Table for access control tools

Other analysis tools	Snyk	Cabal Audit	Calligraphy	Graph Trace	HATS
Approach	Static	Static	Static	Dynamic	Static

(b) Table for other Haskell call graph projects

Table 1: Tables of security tools discussed in section 3 and whether they take a static, dynamic, or hybrid approach.

3.2 Access Control

Access control allows a user to set certain permissions (e.g. read or write access to a file) for an arbitrary piece of code. Many tools, for different languages, already exist, and many take different approaches. Mir is an access control system introduced by Nikos Vasilakis et al.[6] to use for JavaScript libraries. Shill is a secure shell scripting language introduced by Scott Moore et al.[7]. Both are examples of dynamic access control systems, meaning they throw a run-time error when a given permission is violated. This is often not ideal, because the violation may be hidden in some obscure program path. This means it might take very thorough testing before this check happens, and only then does it become clear a permission was violated. Or it goes unnoticed and crashes in the hands of end users. It also introduces overhead due to the dynamic checks.

Some languages, like JavaScript, support language features allowing the code to decide at run-time which code to execute. A JavaScript program could, for example, download source code from the internet, immediately bring it into scope, and call its functions. Or it could dynamically load in a text file containing some code and use the *eval* function to run it. This means that for some languages, static analysis tools cannot provide complete information.

3.2.1 IFC

IFC (Information Flow Control) is a form of access control with a more specific goal. It guarantees secret information won't be leaked when running untrusted code. It generally does this by labelling variables with labels in varying levels of secrecy. The IFC tools we will discuss in this section all come in the form of a Haskell library, which is possible thanks to Haskell's strong and expressive type system.

Deian Stefan et al.[8] implemented a flexible dynamic IFC in the form of a library in Haskell. They allow developers to set secrecy labels at run-time such that secret information cannot be leaked. This means during execution, the system can change whether a certain user is allowed to view some secret information. Because it is dynamic, it is not always optimal (e.g. doing more checks than needed). Pablo Buiras et al.[9] improved this with HLIO by adding some static checks as well, making it hybrid. However, these libraries still have the same drawbacks as the systems in the previous section.

Haskell, thanks to its static nature, can have tools to enforce permissions at compile time. One example of this is the MAC (Mandatory Access Control) library for Haskell[10]. Using this library, developers can annotate variables with labels in their type, indicating their secrecy. The MAC library then promises this variable not to be leaked to an output with a lower secrecy level.

An inherent drawback to these libraries is that they require the untrusted code to be written in a specific monad. This makes these libraries not backwards compatible. HATS works with any Haskell code, no matter how it is written.

3.2.2 Safe Haskell

David Terei et al.[11] took a fully static approach when they introduced a notion of safety in their paper on Safe Haskell. Safe Haskell is a GHC (The Glasgow Haskell Compiler) extension, which controls access to certain unsafe Haskell and GHC features. It guarantees certain safety properties, such as type safety (a compiled program does not end up in an undefined state), referential transparency (pure functions do not have side effects), and other properties. In the following example, module A would

be inferred as unsafe by Safe Haskell, because the module uses RULES, which in this case replace any reference to *foo* with the expression *putStrLn "hello world!"*. This changes the behaviour of the program. Therefore, Safe Haskell does not allow user-defined RULES at all.

```haskell
module A where

{-# RULES "foo" foo = putStrLn "hello world!" #-}
{-# NOINLINE foo #-}
foo :: IO ()
foo = return ()
```

Module B will also be inferred as unsafe, because the function f does not preserve referential transparency.

```haskell
module B where

usec :: IO Integer
usec = getPOSIXTime >>= return . truncate . (1000000 *)

f :: Integer -> Integer
f x = x + unsafePerformIO usec
```

Like Safe Haskell, HATS analyses the program statically. This means it doesn't have to rely on any run-time system. HATS can be used with Safe Haskell to ensure it does not miss any otherwise injected dependencies through unsafe Haskell features.

Safe Haskell has a notion of trust, where a trusted module can use unsafe language features (e.g. for performance reasons). Furthermore, these modules are expected (but not enforced) to export a safe API. When a package author marks a module as *trustworthy*, they take responsibility for it. When we trust such a module, we must trust the author to ensure their dependencies do not break any safety guarantees. This means that we must trust some packages not to inject any dependencies through unsafe features to use them. HATS does not produce false negatives, so long as this trust is not betrayed.

Though HATS does not do any access control directly, it does provide the user with the information they need to infer what access untrusted code requests, by revealing what functions are being called, potentially multiple layers deep in the dependency tree.

3.3 Haskell Call Graph Projects

HATS is not the only tool to generate a call graph of the program, this section discusses a few others.

Cabal Audit[5] Looks at the call graph to infer whether the project depends on a known vulnerable function. This means cabal-audit is also a vulnerability scanner. They experimented with different methods, like core plugins, and loading hi files in a GHC session. However, we had trouble with the latter technique, because GHC wouldn't allow us to load the hi files of hidden modules. HATS is not focused on finding known vulnerabilities, but can instead be used to audit packages for any reason while tracking changes which might reveal new vulnerabilities.

Calligraphy[12] Generates the call graph based on the hie files produced by GHC. It does not allow the user to query the graph, leaving the user with big hard to read graphs.

Graph Trace[13] Made using a source plugin[14]. Generates the graph dynamically as you run the code. They do this to include the argument and return values of the called functions. It does require functions to include a type signature, otherwise these functions can slip under the radar. HATS allows users to audit packages without needing to run them.

When the call graph becomes too big to be useful, there are two options:

- Condense the graph by decreasing the granularity (e.g showing just the modules)

- Filter the graph for specific nodes.

We chose to implement the latter while Calligraphy uses the former. We did this because it makes the graph smaller without losing the information about the program paths. This means we can still run queries to search for them, making it easier to learn and audit the behaviour of a package precisely.

4 Implementation

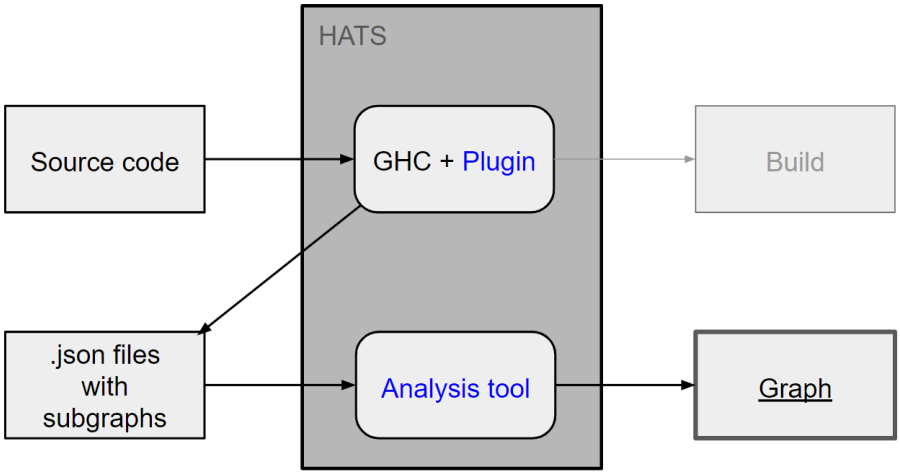


Figure 5: High-level pipeline of the analysis tool. The components of the analysis tool are coloured blue. HATS does not use the resulting build, it is therefore greyed out.

HATS is composed of two parts. One GHC plugin, which turns the program into a graph. And one executable analysing and filtering the graph, as is shown in figure 5. We call the executable the Graph Filter. The reason for HATS consisting of two parts is to prevent doing unnecessary work. To run the plugin, the user must build their program and its dependencies. This can take a rather long time. However, a user might want to filter this graph in multiple ways, to audit different parts of a project. Therefore, Hats generates the call graph once and allows users to reuse it multiple times in the Graph Filter.

4.1 Plugin

To find functional dependencies inside a program, we extend GHC with a core plugin. The GHC pipeline performs many steps to turn source code into an executable or a library. At almost any point in that pipeline, GHC allows a developer to insert an extra piece of logic known as a plugin[14].

4.1.1 Core Background

Near the end of the GHC compilation pipeline, the source code has been simplified, such that it is now part of the Core language. This makes it easier for GHC to run optimisation passes. A Core plugin allows a developer to either read from this core representation of the program or transform it. Figure 6 shows when GHC runs our plugin in its pipeline.

Let's look at an example of what Core looks like by simplifying the following snippet of Haskell code.

```
1 fac :: Int -> Int -> Int
2 fac a 0 = a
3 fac a n =
4     fac (n*a) (n-1)
```

Which translated to Core looks like this:

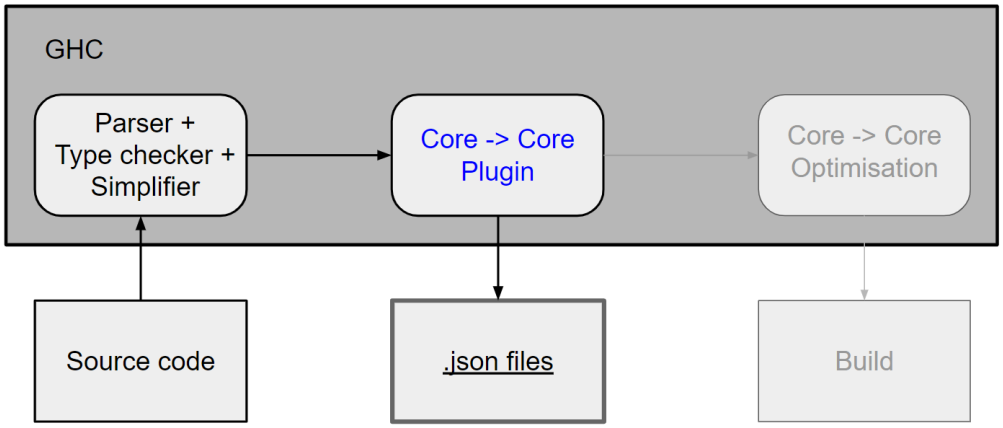


Figure 6: High-level simplified diagram of how GHC uses the plugin component of the analysis tool.

```
fac :: Int -> Int -> Int
fac = \ (a :: Int) (ds :: Int) ->
  case ds of wild { I# ds1 ->
  case ds1 of _ {
    __DEFAULT ->
      fac (* @ Int $fNumInt wild a)
          (- @ Int $fNumInt wild (I# 1));
    0 -> a
    }
    }
```

In Core, polymorphic functions must explicitly be given type arguments. Here *$fNumInt* is a variable referring to the *Num* instance of *Int*, which in this example is passed to the functions $(*)$ and $(-)$ as type class arguments. This also highlights one drawback to analysing the Core representation of the program, as it doesn't quite look like the source code any more. A variable like *$fNumInt* might show up in the final result, which could confuse users unfamiliar with the Core syntax. However, these variables don't show up too often and could be renamed for clarity.

4.1.2 Reading Dependencies from Core

The Core language is, simply put, a list of variable and expression pairs, called Core bindings. HATS reads from these Core bindings, turning each binding into a pair of variables and a list of non-local variables that occur in the expression. In the case of the previous example, this would result in the following list of pairs.

```
[("fac", ["(*)", "(+)", "$fNumInt"])]
```

Along with these variables, our plugin keeps track of which module and which package it comes from and whether it can perform IO actions, this includes FFI (Foreign Function Interface) bindings. When using Safe Haskell, FFI bindings must always be annotated with the IO type. However, HATS captures FFI bindings either way. This whole process translates to the following Haskell type:

```
type CoreBind = (Var, CoreExpr)

data Decl = Decl {
    declOccname :: String,
    declModule  :: String,
    declUnit    :: String,
    declIsIO    :: Bool}

findDependencies :: [CoreBind] -> [(Decl, [Decl])]
```

Finally, our plugin stores this result in a JSON file, one graph for each module, to be later collected by the graph tool.

4.2 Graph Filter

The Graph Filter is the second part of HATS. It starts by combining the sub-graphs from the plugin into one big call graph. Then, it enables users to filter down this graph based on a certain specification.

4.2.1 Filter Specifications

Neo4j already allows the user to query the call graph. However, because it is rather slow with bigger graphs running on a simple laptop, we need to filter down this graph to only the relevant declarations. HATS allows us to do this in a few different ways. The user can specify the result graph to:

- only include paths starting at a group of nodes (called root nodes).
- only include paths ending at a group of nodes (called target nodes).
- only include paths going through a group of nodes (called middle nodes).

Any combination of these three types of filter options is also allowed. For each of these options, the user must provide a group of nodes. Declarations can only be grouped by module or package. It is also possible to add individual declarations to a group.

The filter specification used for Figure 3 translates to the following command[1]:

```
1 $ hats --neo4j --graphviz --json path/to/json/files/ --filterIO --rd main:Main:main
```

4.2.2 Graph Visualisation

HATS currently uses two external data visualisation tools.

- Graph Viz: Easy to use, generates a picture of the graph. The picture can be chaotic for bigger graphs.
- Neo4j: Has an interactive interface and allows the user to query the graph. This is done with Neo4j's DSL (Domain Specific Language) called Cypher.

HATS has two flags for outputting the graph in a format interpretable by these external tools. These formats are languages called Cypher and Dot, which can be interpreted by Neo4j and Graph Viz, respectively. When one of these flags is used, HATS writes the filtered graph to a text file in the correct format. Currently, the graphs it outputs are minimal viable graphs. This means no customisations, such as using colours or grouping nodes in boxes, are used when outputting the graph to these formats, which these DSLs can do.

4.2.3 Tracking Changes

As discussed in section 2.3, HATS can also spit out the list of dependencies of a specific function. When the user specifies to track a function, it will write information about each transitive dependency to a file. More specifically, for each dependency, it writes the function, the function it is referenced in, and how many times it is referenced in that function. An example of this can be found in section 2.3

The user is given 2 options:

- Include every referenced function
- Only include leaf nodes of the call graph

The leaf nodes are IO functions which do not reference any other IO functions. This means these are either primitive functions from the standard libraries[2] (e.g. *readFile*, *print*, etc.) or FFI bindings. These functions define the IO behaviour of a program. However, when only leaf functions are included, it loses some nuances of how they are used.

[1] *rd* stands for root declaration.
[2] Also known as "boot" libraries

5 Using HATS

The source code for HATS is available on GitHub (see [15]). Sadly, running a plugin on every module in every package a project (transitively) depends on can be tricky. First, the plugin must be globally installed. We used Cabal (a package manager for Haskell) for this and configured our project to use the same GHC flags when compiling every package with a *cabal.project* file. These GHC flags must then be set to contain all the information GHC needs to run the plugin. Hopefully, support for running plugins will be better in the future[16]. Cabal is not the only tool to support running plugins, but the others' support is not much, if at all, better than Cabal's.

6 Limitations and Final Remarks

False positives The call graph resulting from HATS might include functions that are not referenced. This is due to type classes not yet having been specialised when our plugin runs. This means that when using one function from a type class, all of them show up. However, methods of specialising type classes exist (like discussed by A. Arvidsson et al.[17]). Or perhaps GHC could introduce a flag for this.

False negatives Because the standard libraries are distributed pre-compiled, HATS might miss some uses of IO functions. Monads which implement the *MonadIO* type class can lift IO functions into themselves. This would normally not matter as HATS would find the uses of these IO functions in their definitions, but that is not possible with pre-compiled libraries. However, every monad with a *MonadIO* instance, must eventually come back to the IO monad for it to run the IO actions, meaning HATS will at least find that such a monad is used.

The same limitation applies to type classes defined in the standard libraries. HATS will only find whether and where they are used, but can't inspect the definitions of the instances.

Furthermore, for this reason, HATS implicitly trusts the standard libraries, as it cannot verify their behaviour by looking at the definitions.

Sadly, the only general fix to these issues is forcing GHC to rebuild the standard libraries. However, this is currently not directly supported by most popular package managers. However, it is possible to do it manually, either by downloading the source code of these libraries or setting it up with more flexible package managers, such as Nix.

Future work The customisations discussed in section 4.2.2 could help the user read the graphs more efficiently.

Tracking value HATS can not track the value of an expression, as this is hard to do statically. This means that if the argument of a function changes, e.g. a string value, it will not show up in the diff tool. If the new argument changes value because it references a different function, the change will be detected.

Language agnostic The Graph Filter is language agnostic, meaning it can be used for any language, as long as it is given the program's call-graph.

References

[1] C. Cimpanu. "Malware found in npm package with millions of weekly downloads." (), [Online]. Available: https://therecord.media/malware-found-in-npm-package-with-millions-of-weekly-downloads. (accessed: 04.June.2024).

[2] P. R. Team. "Phylum discovers sophisticated ongoing attack on npm." (), [Online]. Available: https://blog.phylum.io/sophisticated-ongoing-attack-discovered-on-npm/. (accessed: 11.June.2024).

[3] L. Tal. "What is typosquatting and how typosquatting attacks are responsible for malicious modules in npm." (), [Online]. Available: https://snyk.io/blog/typosquatting-attacks/. (accessed: 18.June.2024).

[4] "Snyk." (), [Online]. Available: https://snyk.io/.

[5] T. Cacqueray. "Cabal-audit." (), [Online]. Available: https://github.com/TristanCacqueray/cabal-audit/.

[6] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Preventing dynamic library compromise on node.js via rwx-based privilege reduction," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1821–1838, ISBN: 9781450384544. DOI: 10.1145/3460120.3484535. [Online]. Available: https://doi.org/10.1145/3460120.3484535.

[7] S. Moore, C. Dimoulas, D. King, and S. Chong, "SHILL: A secure shell scripting language," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 183–199, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/moore.

[8] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," *SIGPLAN Not.*, vol. 46, no. 12, pp. 95–106, Sep. 2011, ISSN: 0362-1340. DOI: 10.1145/2096148.2034688. [Online]. Available: https://doi.org/10.1145/2096148.2034688.

[9] P. Buiras, D. Vytiniotis, and A. Russo, "Hlio: Mixing static and dynamic typing for information-flow control in haskell," *SIGPLAN Not.*, vol. 50, no. 9, pp. 289–301, Aug. 2015, ISSN: 0362-1340. DOI: 10.1145/2858949.2784758. [Online]. Available: https://doi.org/10.1145/2858949.2784758.

[10] A. Russo, "Functional pearl: Two can keep a secret, if one of them uses haskell," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015, Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 280–288, ISBN: 9781450336697. DOI: 10.1145/2784731.2784756. [Online]. Available: https://doi.org/10.1145/2784731.2784756.

[11] D. Terei, D. Mazières, S. Marlow, and S. P. Jones, "Safe haskell," in *Proceedings of the 2012 Haskell Symposium*, 2012.

[12] J. Carpay, Hécate, and T. Cacqueray. "Calligraphy." (), [Online]. Available: https://github.com/jonascarpay/calligraphy.

[13] aaronallen8455. "Graph-trace." (), [Online]. Available: https://github.com/aaronallen8455/graph-trace.

[14] M. Pickering, N. Wu, and B. Németh, "Working with source plugins," in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2019, Berlin, Germany: Association for Computing Machinery, 2019, pp. 85–97, ISBN: 9781450368131. DOI: 10.1145/3331545.3342599. [Online]. Available: https://doi.org/10.1145/3331545.3342599.

[15] T. Blok. "Haskell auditing tool." (), [Online]. Available: https://github.com/TimoBlok/haskell-auditing-tool.

[16] michaelpj. "Cabal issue 7901." (), [Online]. Available: https://github.com/haskell/cabal/issues/7901.

[17] A. Arvidsson, M. Johansson, and R. Touche, "Proving type class laws for haskell," in *Trends in Functional Programming*, D. Van Horn and J. Hughes, Eds., Cham: Springer International Publishing, 2019, pp. 61–74, ISBN: 978-3-030-14805-8.