Cif: Language-based Information-flow Security in C++

Raphael Isemann - Group 26

May 9, 2021

Abstract – This report introduces *Cif*, an extension to the C++ programming language that allows expressing and enforcing information flow inside a program. A program which information flow is annotated via Cif can be automatically and statically verified. Cif is inspired by Jif, which enforces information flow in Java programs. However, Cif was simplified in both usage and implementation and doesn't break compatibility with the ISO C++ standard.

1 Introduction

At the beginning of May 2018, the internet services GitHub [7] and Twitter [8] have revealed that user passwords have unintentionally been written to internal logs. Those passwords were written as clear text, which prompted these services to advice users to change their passwords. As always after these kinds of events the question arises: How can we prevent this in the future?

The underlying cause for these security problems is unrestricted information flow inside a program: Data containing sensitive information can easily be copied to a memory region that contains public information. And inside a single process, neither the computer nor the operating system will block such attempts to declassify information. After all, from the point of view of the computer, all data inside the process was created equal.

A concrete example for such an information flow would be a password string that is passed to a function that appends its argument to an internal log file. Obviously the password is sensitive data and the function should only accept non-sensitive data, but the program will most likely successfully compile and run.

To restrict the information flow accordingly, programmers first need a way to express the intended information flow in their program. They also need a verifier which ensures that their intended information flow can't be unintentionally violated by the program.

This can either be done by utilizing the type system rules of a programming language or by extending the language with information flow-specific constructs. In this report we present our implementation of the latter option for the C++ programming language: *Cif*.

2 Background: Jif for Java

The work behind this report was inspired by Jif[1][2], which brings information flow security to the Java programming language. Jif is a language extension for Java that allows adding labels to variable declarations. These labels describe what information flow these variable allows, i.e. who can read their value and write new values to them.

We see in Listing 1 an example program that illustrates Jif's labels. We declare a variable x that is owned by the principal 'A' and can be read by principal 'B'. We also declare a variable y that is also controlled by the prin-

```
1 int {A->B} x;
2 int {A->B, C} y;
3 x = y; // Allowed flow.
4 y = x; // error: Flow not allowed.
```

Listing 1: An example for Jif code with annotated. Adapted from the Jif reference manual examples [4].

cipal ’A’ and can be read by the principals ’B’ and ’C’.

Jif now allows writing ’y’ to ’x’ in line 3, as no additional principal gains implicit read accesses by this. However, writing ’x’ to ’y’ is forbidden as principal ’C’, who previously couldn’t read the value of ’x’, now gains access to the value of ’x’ as it has been copied to ’y’ (which ’C’ can read).

If the programmer wants to signal to Jif that line 4’s flow is permitted, the ’x’ value would need to be declassified with Jif’s *declassify* method. This method takes a value, an old label and a new label. The value is then checked to match the old label and is then assigned the new label. By forcing the programmer to explicitly state from which source to which target principal the information is flowing, unintentional declassification is less likely to occur.

Jif also has more complex features such as run-time label checking, principal hierarchies and automatic label inference. However, for simple programs, labels and declassification are enough to describe their information flow.

3 Introducing Cif

Cif brings information flow security to C++ in the same way as Jif: It adds language extensions for annotating the information flow in a program and provides a verifier to enforce this flow. The goal is to make Cif a tool for software engineers to secure their own software against unintentional information leaks. Cif is not intended to secure or contain programs written by untrusted parties.

Cif also aims to be a practical solution to information flow security. This means that it aims to be simple to understand for users and adding Cif to a program should not require drastic changes to the software system itself.

To achieve this, Cif avoids some of the complexity and incompatibilities that one would encounter with Jif:

First of all, Cif programs are still valid C++ code. All annotations that can be added to programs can be correctly parsed by standard C++ compilers, even though only Cif-aware compilers enforce the information flow by using those annotations. This means that adding Cif to a project does not require changing the normal build process. It also means that all tools that could be used without Cif (i.e. Integrated Development Environments, source code formatters and other analyzers) can continued to be used with Cif annotations in the source code.

Secondly, Cif attempts less complex than Jif. There is no hierarchy of principals that can act for each other and each variable is only labelled by the list of owners. For this reason, Cif also does not distinguish between read and write access to a certain, even though it can be simulated to a certain degree by qualifying types with *const*.

Listing 2 shows a simple authentication program annotated with Cif. The `log` function here takes a string that should be written to a log file and the `login` handles the user password and logs the login attempt. The only parts of the source code that are related to Cif are the additional `include` directive for the header `CIF.h`, a `CIFPureList` declaration and the `CIFLabel` annotations in the `login` and `sha256` function signatures.

A `CIFLabel` annotation marks in Cif that the associated variable is owned by a certain

```
 1 #include <string>
 2 #include <CIF.h>
 3
 4 using namespace std;
 5 CIFPureList { using string; }
 6
 7 void log(string Msg);
 8
 9 string getPWHash(string User);
10 string sha256(CIFLabel("Secret")string);
11
12 bool login(string UserName,
13            CIFLabel("Secret") string Password) {
14   string PWHash = sha256(Password);
15   log("Login: " + Username + ":" + Password);
16   return PWHash == getPWHash(UserName);;
17 }
```

Listing 2: A basic C++ program using Cif

entity. In this case the `Secret` entity owns the data to designate that the values should not be known by anyone. The `login` and `sha256` functions both take values that are owned by `Secret`. The `log` function however can not accept data owned by `Secret`.

As we did in the Jif example, we will also violate the information flow in this example to demonstrate the verifier: In line 15 we concatenate the username and the associated password into a string which is then passed to the `log` function. While concatenating the string itself is not a violation, the resulting string containing both username and password will still be classified as being owned by `Secret`. Then this string is passed to the `log` function which is not allowed to accept `Secret` data.

This information flow violation is similar to the violations discussed in the introduction. However, with the additional annotations we placed in the program, we are now able to detect it.

4 The Cif verifier

Like Jif, Cif also features an automatic verifier that will report the violating from listing 2. The Cif verifier consists of two parts:

The first part is the Cif header `CIF.h` which needs to be included in each source file that uses Cif annotations. This header provides a set of preprocessor macros that can be used to express the security classes of variables. These preprocessor macros are implemented in a way that they will be translated into an empty token sequence on all non-Cif compatible compilers and are therefore invisible for most compilers. When verifying the source code with Cif however, the macros are translated into meaningful annotations (mostly clang's `__attribute__((annotate("value")))` annotations, which don't influence the behavior of the code).

The second part is a plugin for Clang's static analyzer infrastructure. This plugin reads the source code and verifies it according to Cif's information flow rules. Any detected violation will be reported to the user. The information about the owners and flow semantics of the code are extracted from the annotations placed by the macros from the `CIF.h` header.

The advantages of using Clang as a basis for the verifier are numerous: Clang offers support for many different dialects and versions of C++. It also simplifies extending Cif to the other languages Clang supports, such as C, Obj-C, Obj-C++ and OpenCL. Furthermore Clang allows the generation of expressive diagnostics for found problems, which the Cif verifier uses to describe information flow violations in an expressive manner to the user.

5 Information flow rules of Cif

When Cif verifies a program, it follows certain rules to decide what information flows are allowed. In general these rules follow the principle that information can always be implicitly classified, but declassification has to be explicitly requested by the programmer.

As it is not a trivial task to verify all flows in a program as a whole, Cif instead models its information flow policy with a lattice structure as described by Denning & Denning [3]. This allows Cif to verify the correctness of the whole program by detecting and verifying only the direct information flows in a program.

5.1 Information flow model

The information flow model of Cif is defined after Denning & Denning [3] as:

$$FM = \langle N, P, SC, \oplus, \rightarrow \rangle$$

We define our logical storage objects N as the variables in our program. Our set of processes P is just a single process as Cif does not enforce any security classes during run time. For any static analysis purposes this single process is enough.

Our set of security classes SC is user defined in the program. All used security classes are a member of SC. Given that our security model is based on this finite set of defined security classes, we can follow Denning & Denning's proposed information flow model for finite sets [3]:

We define that all combinations of these labels are part of SC. Additionally, we define an empty set of security classes L as the lower bound of SC. The class-combining operator $\oplus$ is defined in Cif as the union of the two given sets of security classes, e.g. $\{A, B\} \oplus \{B, C\} = \{A, B, C\}$. The flow relation $\rightarrow$ defines for which pair of security classes a flow is permitted. Cif only allows flows $A \rightarrow B$ for which $A \subseteq B$ is true which is also following Denning & Denning's model.

With this lattice model in place, we can now verify the information flow policy of the whole program if we show that all direct flows in the program are following this policy [3]. The only remaining task is now to detect and model all explicit and implicit flows that can be expressed in C++.

5.2 Explicit Flows

Explicit flows (as defined by Denning & Denning [3]) happen in C++ on assignments, field initializers and member initializer lists and function calls. For all these explicit flows we have a target variable declaration to which the information flows and a source expression from which the information flows.

When evaluating the source expression, Cif is implicitly calculating the least upper bound ($\oplus$) of all security classes used in the expression. In the example below, we have an expression that performs an addition on the classes $\{A, C\}$ and $\{B\}$.

```
CIFLabel("A,C") int a = 1;
CIFLabel("B") int b = 3;
CIFLabel("A,B,C") int c = a + b;
```

After determining the security class of the expression, Cif validates the flow by checking if the set of source security classes is a subset of the target security classes. If this condition is fulfilled, the flow is allowed. If not, the flow will be reported to the user and the program will be rejected.

The empty set of security classes L is in Cif defined for any unlabeled variables and expressions. L is also the default security class of all language constructs holding information (e.g. constants and other literals) as motivated in [3, p. 505]. As any information flows inside L are allowed, this means Cif accepts standard C++ source code without reporting failures. However, as Cif does not allow implicit declassification to L and automatically classifies results leaving L, this does not violate the information flow policy of the whole program.

5.3 Functions

Functions in Cif are handled in a special way. Cif considers them as gateways to unknown code and uses only the signature to verify the flow to and from them. When calling a function, Cif verifies that the passed values match to the security classes of the parameters. Cif also once verifies the function body that it used the parameters in a valid way. Given that our rule system based on transitive rules, this ensures that the overall information flow over function boundaries is not violating the information flow model.

Functions that are implemented in another file are also verified over their signature. Cif assumes that the implementation of a labeled function was also verified, but it can't actually check if it's assumption is true. This is because the function implementation may reside in a external compiled library and its source code is not available. As the function may be compiled without being verified by Cif, it is possible that the function body violates Cif's information flow rules.

A special case are pure functions. If a function is marked as pure, then Cif models the function call as a side effect free computation on the operands. This means that no labels will be check on the operands that pass into the function, but the return value of the function has the least restrictive security class that can hold the values of all operands.

5.4 Implicit Flows

Implicit flows in Cif occur when the control flow is influenced by the value of classified data. `if` statements for example are recognized as implicit flows in Cif, where the implicit flow is based on the class inside the condition expression of the `if` statement. The same applies for `while` and `for` loops. If the condition expression has security class that is not L, then the *security context* of the 'then' and 'else' cases (or respectively the loop body) is merged with the security class of the condition expression.

The security context of a statement describes in Cif additional security classes that are applied to every read value. A security context is always inherited to all child statements and can never be declassified (i.e. once a security context gains the security class 'A', every read value in all child statements will be also classified with 'A'). However, a security context can be further classified for example by multiple nested `if` statements.

In the example below we see an implicit flow because we branch a value of the security class A. A will be added to the security context of the statement in the 'then' branch, which causes that the read value `1` is now classified by A. `1` usually has the empty security class L because it is a constant, but Cif will now also calculate the least upper bound with the security context class A to get the final security class of this expression ($L \oplus A = A$).

```
CIFLabel("A") bool a = 1;
bool b = 0;
if (a) {
  b = 1; // Flow violation, '1' owned by 'A'.
}
```

The implicit flow can further influence the security context of the parent statements if any of the branches can cause the execution of a `return` statement. As `return` will terminate the current execution in the function, the execution of all statements following the if statement also depends on the value of the condition expression.

Cif models these extended implicit flows by also adding the security class of the condition to all parent context if any of the children could call return.

In the example below we see code that leaks the value of `a` to `b` by returning if `a` is true. When Cif discovers that the `if` statement can return, it also adds the security class of the condition variable to the parent context. Now the `0` literal is owned by the same security class and can not by anymore assigned to `b` without declassifying it explicitly.

```
void hoge() {
  CIFLabel("A") bool a = 1;
  bool b = 1;
  if (a) {
    return;
  }
  b = 0; // Flow violation, '0' owned by 'A'.
  print(b);
}
```

One language construct that can create implicit flows is however not supported by Cif: `goto`. While `goto` is still supported in C++, it is also barely used in modern code and hard to integrate into our information flow model. For this reason Cif is silently ignoring any `goto` statement it encounters in the source code.

5.5 Structures

C++ supports classes and structures. Handling these in an convenient way in Cif requires new rules for these language features.

Structs and all derived types (e.g. pointer and references to structs) are treated like variables. The user can label them and all code involving a labelled structure type such as a member access or taking the address behave in the same way as accessing a variable (see listing 3).

```
struct Str {
  char *data;
  unsigned len;
};

CIFLabel("A") Str aStr;
// These all violate flow from A to L
Str bStr = AStr;
unsigned len = AStr.len;
Str *ptr = &aStr;
```

Listing 3: Structs in Cif.

```
struct User {
  Str name;
  CIFLabel("Secret") Str passwd;
};
CIFLabel("Medical") Str aStr;
// Secret and Medical get merged.
CIFLabel("Medical,Secret") Str P = aStr.pass;
```

Listing 4: Label merging and structs in Cif.

The user can also specify additional labels for certain members. These labels are merged with the label that the user has supplied for the variable (see listing 4).

5.6 Classes

Classes are more complex than structures, and therefore also have a set of more complex rules to ensure information flow security.

The biggest change is that class variables can't be marked with Cif labels. This is caused by the fact that classes contain functions that can reference the current object, which should be affected by the label on the class declaration. This means that all functions would need to be verified again by Cif every time a label is used on a class.

This verification however is often not pos-

```
// File: String.h
class String {
  const char *data;
public:
  const char *get();
};
// File: String.cpp
void Foo::get(const char **out) {
  *out = data;
}
// other source file:
CIFLabel("Password") String pw;
const char *public;
// Cif can't detect this violation.
pw.get(&public);
```

Listing 5: Theoretical violation of the information policy with labelled classes. Note that this is not a real Cif program, but only for demonstration purposes.

sible as the source code for the function body is usually not in the same translation as the code that is using the class. An example for such a situation is implementing a class in a source file while declaring it in a header. Every user of the header now lacks the function bodies, which causes that Cif is unable to verify them for the new label. In the worst case a function can leak information this way as in the example in listing 5.

Classes have however other ways to be annotated with Cif labels. The most obvious way to label a class is to label the class declaration itself. Any security class attached to a class will be automatically propagated to all members and methods of this class (see listing 6). It will also propagated to all subclasses and classify their members and methods. The possibility to annotate certain members themselves also continues to be allowed with classes and works in the same way as when used with struct members.

```
class CIFLabel("A") A {
public:
  CIFLabel("B") int foo() {
    return 2;
  }
};

void foo() {
  A I;
  // Violates from from A,B to L
  int x = I.foo();
}
```

Listing 6: Example of using Cif with classes.

```
#include <vector>
CIFPureList { using std::vector }
void foo() {
  CIFLabel("Secret") std::vector<int> secretPrimes;
  CIFLabel("Users") std::vector<int> userIds;
  // Disallowed flow from Users to L.
  int i = userIds.front();
}
```

Listing 7: Containers in Cif.

5.7 Containers

Containers are special classes that were marked by the user via `CIFPureList`. These classes can - unlike normal classes - be marked when used in a variable similar with a security class. The user states via the CIFPureList declaration that these classes only operate on its contained data and only return data via its return types.

Classes marked as containers behave like structures, meaning that all member access and other operations on the container variable itself will have the same security class as the container. Calling any member method of a container will return a result with the container security class. Also, all parameters of the member methods have the security class of the container as demonstrated in listing 7.

6 Cif's Syntax

Cif's syntax consists of four additional language constructs.

The most important construct are labels which designate the security class of an variable. They can be used for any variable declaration. They can also be used on function declarations, where they attach to the return value of the function.

A label consists of a call to the `CIFLabel` macro and a string containing the list of security classes. Multiple security classes can be specified and are separated by comma. Security classes are case sensitive and can contain any alphanumeric character.

In the example below we describe the information flow in the function `foo`. The return value of `foo` has the security class `Public`. The only argument has the security class `Secret` and the variable `Result` has the two security classes `Public,Secret`.

```
CIFLabel("Public")
int foo(CIFLabel("Secret") char * PW) {
  CIFLabel("Public,Secret") int Result;
  Result = strcmp(PW, "letmein");
  return Result;
}
```

The second construct is the declassify function. Like in Jif, it takes a value, the expected security classes and the target security classes. The declassify function verifies that the value has the expected security class and then changes it's classification to the target security class.

In the example in listing 8 we declassify the `PW` parameter from it's expected class `Secret` to an empty security class.

The third and fourth construct are the `CIFPure` (and `CIFPureList`) declarations. These declarations exist for practical reasons, as many existing functions are not annotated

```
void login(CIFLabel("Secret") char *PW) {
  checkPW(CIFDeclassify("Secret->", PW));
  // ...
}
```

Listing 8: Declassifying in Cif

but can be considered as pure. These functions don't store any information themselves but only return a value computed exclusively from the input values.

In Cif these functions are modeled by allowing everyone to pass any kind of value to them. However, the result of these functions is then an union of all security classes of all arguments passed to the function call.

```
// Marks function as pure.
CIFPure void foo();
// Marks multiple functions as pure.
CIFPureList {
  using ::strcmp;
  using ::strlen;
}
CIFLabel("A,B")
int foo(CIFLabel("A") char *a,
         CIFLabel("B") char *b) {
  return strcmp(a, b);
}
```

`CIFPure` is also used to mark classes as containers and follows the same pattern. When using `CIFPure` on a template, it applies to all template specializations and instanziations of a given template.

6.1 Labeling declarations in C++

In C++ a single function or class may be declared multiple times in a single translation unit. For Cif this is both an opportunity and a challenge:

Redeclarations allow the user to add Cif labels to existing declarations. For this the user redeclares the declaration and then adds the related labels to the redeclaration. When annotating an existing external library with Cif, this is less intrusive to the programming workflow than creating a correctly labeled wrapper function for each library function.

However, Cif now has the challenge to manage the labels in the different redeclarations a user may provide. It's also not clear how exactly labeled redeclarations affect existing labels. Cif could either decide that only the latest redeclaration labels define the information flow (and previous labels are ignored), or that the sum of all labels from all redeclarations define the correct information flow.

The actual implementation in Cif is that all labels from all redeclarations are merged accordingly. The reasoning behind this behavior is that it follows the closest the principle that classification should be implicit and declassification explicit. If we would allow Cif to ignore existing labels implicitly when we redeclare a function, we could for example accidentally declassify a function by redeclaring it with no labels.

7 Limitations and unsupported features

Cif can't guarantee that all flows follow the rules of the information flow model. These undetected flows can be based on side channels or unsupported language features.

7.1 Side channels

Side channels are hard to detect when enforcing information flow security. Possible side channels in C++ are numerous as it's possible to access the hardware directly from the code. Example are timing based side channels (e.g. waiting for a certain amount of time), influencing battery or CPU usage by performing certain calculations or any kind of IO operations such as networking. Cif does not enforce information flows using these side channels, as doing so is very hard for a verifier that has no access to the run time information these side channels utilize.

7.2 Assembly code

Cif does not enforce information flow in inlined assembly code. The reason for this is that it would require an unreasonable amount of work for such a rarely used feature. Especially the understanding of different hardware architectures and assembly languages would require to develop a completely new verifier that then needs to be integrated into the C++ verifier.

7.3 Exceptions

Exceptions in C++ do not have to be declared in the signature of a function that throws them, which means that Cif is not aware whether a called function throws an exception. Cif also can't track where a thrown exception will be caught, meaning that exceptions fully circumvent the Cif information flow model.

As Cif has no run time information, it therefore can't handle exceptions in any way. This is problematic as it allows the undetected violation of the information flow policy.

The best heuristic to simulate this behavior is to use the symbolic execution capabilities of Clang to detect where exception could be thrown and caught. However, this approach is also very limited as often exception are caught and thrown in different translation units. As Cif behaves like a compiler (which only sees a single translation unit at once), it can not detect any exceptions crossing translation unit boundaries.

As there is no full solution to this problem, Cif currently just ignores exceptions when verifying source code.

8 Using Cif in practice

As Cif is based on clang, it integrates into every program that is already integrating with the Clang static analyzer. Examples tools that offer this integration are clang's own *scan-build* tool, the QtCreator IDE and Apple's Xcode IDE.

Cif is intended to be used as a custom Clang binary. After building this custom clang binary with Cif support, it can be used in the same place as a real clang binary for analyzing source code. Both QtCreator and XCode accept Cif as a custom clang that then is used inside the IDE to analyze the source code. Also, with the `--analyze` flag of clang the checker can be run directly on source code.

When integrating a project into Cif, it's also necessary to ship the `CIF.h` header in some way. The easiest solution is to copy the header into the project itself. With this no user has to install the `CIF.h` header first to compile the project. However, the disadvantage is that this header needs to be updated if a different Cif version is used to analyze the project.

8.1 Building Cif

Cif is build as a part of Clang, which means that it is necessary to compile Clang + LLVM to use Cif. First LLVM needs to be setup by getting the source code for revision 333167. Then in the `tools` directory of the LLVM source code the Cif source code (available here [5]) needs to be placed under the name `clang`.

Now it's just a matter of creating an empty build directory and running from the command line in the build directory `cmake /path/to/llvm/src/dir ; make check-clang`. This will build LLVM/Clang and runs all the tests inside Clang (which include the Cif tests).

8.2 Example projects

To test how Cif performs on real software, we used Cif on widespread C software and annotated the information flow in their password handling code. The selected projects were OpenSSH, Apache HTTP Server Project, the nginx web server and Lighttpd. For OpenSSH we handled the password authentication module, while for the different webserver we handled the HTTP basic authentication backend. The annotated source code of these projects is available here [6].

It should be noted that all tested software was written in C while Cif was clearly intended to be mainly used on C++. The reason for this is that most of the open-source software we found and that was relevant (i.e. had custom password handling code) was written in C.

To integrate Cif into the build systems of the different projects, we first installed the `CIF.h` header into the system include directory. We could have also copied it into the different projects, but providing the header from a single include directory simplifies upgrading/changing the header during development.

The actual setup of the Cif verifier was also trivial: Clang's scan-build tool could automatically hook into the different build systems and required no changes to the build files themselves.

As scan-build automatically runs multiple analyzers on the build software, we also disabled all checker beside the Cif verifier in scan-build to make the output more readable (see the `scan.sh` in the examples repository[6]).

After successfully integrating Cif, we started annotating the source code by looking for variables that we assumed to contain passwords. Once we found a variable that most likely contained a password, we attached a Cif label to it. Every time we attached a label to a password variable, we verified the project with Cif and then followed the information flow violations that Cif found in our incompletely annotated source code.

Depending on the analyzed software, this strategy lead to different results:

For the OpenSSH project, we eventually reached the point where OpenSSH passes the password to the respective backend service (e.g. PAM). As thse backends are not part of the compiled project (which means that Cif can't verify their already compiled source code), we decided to declassify the password as soon as we reached this point. It should be mentioned that it would be possible to also build the specific backend alongside OpenSSH to completely model the information flow in OpenSSH.

For the different password handling servers we finalized the model when we reached the final password comparison code or the point where the password was forwarded into a hash function. In both cases considered the result of this operation to be no longer sensitive data and only verified with Cif that the password did not leak anywhere else.

When implementing Cif-compability, we mostly had to extend the source code with annotations to declassify the password to forward it to a harmless check that didn't leak any significant information (i.e. checking if the password is empty). Otherwise the passwords were usually just forwarded to another function that either was implemented in external code (e.g. because they are from an authentication backend) or that needed to be annotated to accept a password.

One suspicious Cif violation was found in nginx: Here we annotated the parameter called `passwd` in the function `ngx_http_auth_basic_crypt_handler`. After annotating this function, Cif noted that we pass this variable to a logging function as seen in listing 9. However, this variable seems to only contain a hashed password. While Cif was also able to identify the information leak, the actual leaked information was no longer sensitive as we wrongly labelled the `passwd` variable.

```
ngx_log_debug3(
  NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
  "rc: %i user: \"%V\" salt: \"%s\"",
  rc, &r->headers_in.user, passwd->data);
```

Listing 9: Suspicious code in nginx.

8.3 Possible future improvements

When using Cif on public projects, we found that annotating utility functions (e.g. checking if a string is empty) was a reoccurring task. While Cif is able to improve this situation by marking the builtin string functions as pure, it should be noted that often projects (especially written in C) implement their own custom string functions. These come with their own special behaviors and optimizations. So to fully resolve this issue, Cif would need to be able to recognize and automatically model these data structures and functions. For projects in C++ this task is easier for the user, as the data structure and functions in a class are already bound to each other, meaning that simply marking the class as pure resolves this issue.

There is also an open question regarding function declarations and their labels. Cif allows functions to be labelled in redeclarations. This is (as discussed above) a useful feature. However, it could leak to unintentional declassification:

For example, we assume that an internal function A is declared in some header. The user now redeclares this function and assigns the return type a new security class. However, another source file might not see this redeclaration and verifies the function body only accordingly to the labels in the original declarations. Obviously in this scenario the developer is at fault, as he redeclared and relabelled his own internal function, but still Cif would fail to recognize this inconsistency as it only sees one translation unit at once.

A solution to this problem would be another smaller verifier that creates a database of all functions and their final security classes in each source file. Then this database could be checked for differences in the security classes between translation units that use this function and translation units that implement this function.

9 Conclusions

Information flow security is important in all projects that handle any kind of sensitive data. Especially leaking user information can have severe consequences for the institutions and companies running the software. Preventing these leaks is therefore more important than ever.

To bring defense mechanisms against these kind of issues to the world of C++ programming, the developed tools need to adapt to the standards and complexity of the ecosystems. C++ is a standardized language with a focus on not making any assumption about the machine or environment in which it is running. C++ tooling should therefore also not deviate from the C++ standard by adding incompatible extensions. It also should not rely on the assumption that some code is compiled with a certain compiler or on a certain operation system.

With these considerations, enforcing information flow security in C++ seems to be best approached via a static tool. And as Cif shows, the necessary infrastructure to implement these tools as static checks exist in the C++ ecosystem. The only task that is still open is developing and using tools like Cif in C++ software systems.

References

[1] Pullicino, K., 2014. Jif: Language-based Information-flow Security in Java.

[2] Andrew C. Myers, January 1999. JFlow: Practical mostly-static information flow control, In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL), pages 228–241, San Antonio, TX.

[3] Denning, D.E. and Denning, P.J., 1977. Certification of programs for secure information flow. Communications of the ACM, 20(7), pp.504-513.

[4] The Jif 3.3 reference manual, `https://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html`

[5] Raphael Isemann, 25.5.2018, Cif GitHub respository, `https://github.com/Teemperor/Cif`

[6] Raphael Isemann, 25.5.2018, Cif examples GitHub respository, `https://github.com/Teemperor/CifExamples`

[7] Zack Whittaker, 25.5.2018, GitHub says bug exposed some plaintext passwords, `https://www.zdnet.com/article/github-says-bug-exposed-account-passwords/`

[8] Parag Agrawal, 25.5.2018, Keeping your account secure, `https://blog.twitter.com/official/en_us/topics/company/2018/keeping-your-account-secure.html`

10 Appendix

The following pages list the source code of Cif. This includes the code for the Clang static analyzer checker, all tests and a patch that integrates the checker into Clang. The file `SecureInformationFlow.cpp` contains the checker and needs to be placed in the folder `lib/StaticAnalyzer/Checkers/` inside the clang source tree. The tests can be placed anywhere in `test/` and the provided patch file needs to be applied to clang itself. While this works, it is recommended to instead use the github repository as explained in the report to download and build Cif.