Diploma Thesis

Tool Support for Statically Checking Confidentiality of Kernel Code

Benjamin Engel
benjamin.engel@inf.tu-dresden.de

July 2008

Dresden University of Technology, Dresden, Germany
Faculty of Computer Science
Institute for System Architecture
Chair for Operating Systems

Supervisors: Prof. Dr. rer. nat. Hermann Härtig
Dipl.-Inform. Marcus Völp

Contents

List of Figures

Acknowledgements

First and foremost I have to thank my parents for their abiding support. They encouraged me to pursuit my goals and put aside their own interests. Most of what I am now I am due to them.

Second, I would like to thank my supervisor Marcus Völp for his guidance and for patiently listening to all my questions. He gave me a lot of freedom while working on my thesis and I am grateful for the many fruitful discussions.

Many thanks to Prof. Hermann Härtig who gave me the opportunity to write this thesis and to all the people of the Operating Systems Group at TU Dresden for their support and their valuable advices.

Thanks to all.

Abstract

Showing automatically that a program preserves the confidentiality of the data it works on improves the confidence in it and increases security. The broader scope of this thesis its to prove non-interference of C++ kernel code. Therefore we developed in this thesis a tool that allows us to prove non-interference of C++ kernel code by checking its information-flow security. We use a flow-sensitive approach to infer data dependencies for a rich subset of C++. Among the supported features are function calls, objects, static variables, references and Java-style pointers. This allows us to automatically analyze a variety of C++ code, while only small manual modifications of the source code are necessary. We facilitate an abstract memory model to represent the state of the program and show that data flowing between statements solely flows through this memory model. To show the feasibility of our approach we analyzed an optimized path of a system call in the Fiasco microkernel, namely the `ipc_short_cut()`*. The detected information flow is conform to the L4 Reference Manual.*

Chapter 1

Introduction

Statically checking program properties is a promising approach to make them less error-prone and more secure. Let the computer do the checking is a feasible way do deal with increasing program sizes and complexity. One of the interesting properties is non-interference. Assume the data that a system works with is classified and different security classes, for instance *high* (confidential) and *low* (public). Non-interference means that an observer with *low* clearance (has access to *low* classified data only), cannot distinguish the output of two program runs with different high input data. In other words: No *high* classified information leaks to *low* classified output variables. In particular, safety- and security-critical systems require such guarantees.

How data is allowed to be used, who should have access (read/write) to it and which data flow is illegal is governed by an *information-flow policy*. It states who is allowed to communicate what to whom.

Being able to precisely analyze where data enters a system, how it gets modified and where it spreads enables proofs that a given program does not *leak* any information to unauthorized entities (persons, programs, etc.) and therefore allows a proof of non-interference. Therefore, the overall question is: ***Where does data come from and where does it go?***

We are interested in information-flow security of kernel code and developed a tool which analyzes information flow in C++ kernel code and C++ programs in general. We demonstrate its feasibility by automatically evaluating a system call of the Fiasco microkernel [Lie95] and show that the detected information flow is conform to the L4 Reference Manual [Lie96].

Other approaches address the problem of statically checking information flow in programs as well, for example JFlow [Mye99] and Jif (Java + information flow, see Section 2.3). They enrich the Java language with annotations describing information-flow policies about who is allowed to access which information. Java has some interesting features that contribute to the security of Java programs, like strong data typing and automatic garbage collection (avoiding memory corruption by pointer misuse), to name only some of them. Despite these features which improve the security of Java programs in general, Java is not designed to be used in operating systems in the first place.

Closer to our work is CQual [FTA02], a framework designed to automatically infer user defined types added to the C language (see Section 2.4). One application of CQual is to automatically add the `const` modifier wherever this

is possible. The programmer adds some `const` modifiers at central points and CQual infers constness of many other expressions [FFA99].

The framework has been rewritten to support C++, hence named CQual++ or Oink [DSWM], making it feasible for the Fiasco microkernel. Unfortunately, CQual/Oink targets slightly different problems than we do. For instance, their approach is not flow-sensitive (they do not consider flow of control) and handles pointer differently. Since we want to analyze kernel code, pointers play an important role. In the presence of pointers, flow-sensitive analysis is known to be more precise [Wu08].

Consequently we have taken the practical ideas from JFlow and CQual/Oink and combined them with the theoretical work on abstract code interpretation [JPW05]. We developed a tool that tracks data flow of kernel code written in a rich subset of C++. Among the supported features are (member) functions with default parameters, instance sensitive object modelling (objects of the same class are treated individually and are not folded all into a single object sharing the same memory region), pointers and references, though no pointer arithmetic yet.

We show the feasibility of our approach by applying it to an optimized path of a system call of the Fiasco microkernel, namely the `ipc_short_cut()`. It sums up to about 700 lines of code, consisting of about 20 classes. We were able to show that the information flow detected equals the flow one would expect: data is sent from one thread to the other, no more information is leaked between them and the data does not leak to any other than the intended locations.

To explain the theoretical background of our approach we need some type system theory, given in the first part of Chapter 2. We compare flow-insensitive and flow-sensitive type systems and give reasons why we have chosen the latter approach. The second part of Chapter 2 relates our work to the alternative approaches mentioned above (JFlow, CQual, CCured) and lines out which of their ideas we applied. Chapter 3 presents our design. Starting with an introduction to abstract syntax trees we explain the memory model we use to reflect state changes. This chapter closes with a discussion of the model for function invocations. After the design, we pick up some interesting implementation issues (`if`∼`then`∼`else` and accesses to our abstract memory model) and explain them in detail in Chapter 4. Chapter 5 gives a step-by-step evaluation, beginning with tiny C++ code snippets like assignments, function invocation, references and information flow between objects and member functions. The chapter concludes with our case study of the `ipc_short_cut()` function. Here we show how information flows during the transfer of data from a sender thread to a receiver thread. Our conclusions and topics left for future work are presented in Chapter 6.

Chapter 2

Background and Related Work

We use use a flow-sensitive type system similar to the dynamic labeling idea formulated by Warnier [JPW05] to achieve information flow security by tracking data flow. Warnier's work is strictly speaking not about flow-sensitive type systems. With dynamic labeling we abstractly interpret a program and verify *afterwards* that the security levels of all its variables have not increased (decreasingness), whereas with flow-sensitive type systems this property is checked each time a rule is applied, i.e., *during* type checking or type inferring.

First, we briefly introduce type systems in general. We then compare flow-sensitive and flow-insensitive type systems with respect to information-flow security. The interested reader finds comprehensive overview on security type systems in the work by Sabelfeld and Myers [SM03].

2.1 Type Systems

Type systems are suitable for information-flow analysis, because they can be used to reason about programs [VS97]. If the applied rules are sound, it is possible to prove properties like non-interference.

Type systems in regular programming languages assign types to variables and expressions. The most common example of a type system is the data type system of languages like C, Java and Pascal. It defines how to interpret a specific bit pattern, which operations are allowed and assures data type compatibility. For example, a 32 bit integer has a completely different meaning from the same 32 bits interpreted as a pointer. Given the basic understanding, the following should not type check:

```
int i = 0;
float f = i;
```

In this example, the variable `i` has type integer and `f` has type float. A simple analysis would not be able to type the second statement. In practice, however, this particular example is accepted in C, Java and Pascal. These languages have rules for automatic type conversion. In this case converting an integer to a float is safe as long as the integer is not near `INT_MIN` or `INT_MAX`. (Otherwise the

32-bit int would no longer fit into a 32-bit float, some bits are required for the exponent). If we were to assign a float to an integer, we would get compiler errors in all three languages.

Data type systems are the most common example of type systems. Assume two integer variables, `a` and `b`, both are annotated with *int*. In C we would write `int a,b;` to declare these two variables and specify that their type is *int*. Adding the two variables should also result in the type *int*, in other words the expression `a + b` should be typed as *int* as well.

There are two slightly different approaches, we can either *type check* an expression or we can *infer* its type. Whit *type checking*, we have to know the type of all expression in advance and prove the soundness of their types by applying typing rules. *Type inference* deduces the types of variables and expressions. This means that at the beginning some, but not all expressions and variables are explicitly typed. By applying *type inference rules* to them we try to deduce the types of the yet unknown expressions. If this can be done conflict free, then the expressions and variables are typeable.

Besides data type systems, although being the most common application, type systems are also used in other areas, including:

- secure information flow to statically check non-interference [Smi01].
- automatic constness inference in order to relieve the programmer from the cumbersome task to manually put `const` wherever it is possible [FFA99]; and,
- format string vulnerabilities: Sometimes untrusted (tainted) data, e.g. entered by a user or received from the network, is used at a place where only trusted(untainted) data should be used. To track if some data that is `tained` is used as `untained` helps to detect these bugs [CW07];

Let's assume we are no longer interested in the data type of a variable, but in its security clearance. In particular we want to enforce that no *high* (confidential) classified information is leaked to *low* (public) classified variables. We assume that only low-classified variables are visible to a *low* observer. In this case we can assign a security level to each variable describing its clearance. This is sufficient to allow the verification that no *high* classified values are written to *low* classified variables, otherwise information would leak. The following example shows how this policy could be violated (the variable `low` has *low* clearance, `high` has *high* clearance):

```
int low,high;
...
low = high;
```

As an observer with *low* security clearance can read `low` and the value of `high` just has been copied to `low`, he can gain access to information that he is not allowed to access. By encoding the security clearance in a type system this illegal assignment would not be typeable. If all typing rules are sound and if a program is type checked, then it is proven that no information leakage exists with respect to the applied rules. The problem then becomes choosing the appropriate rules. This choice is typically a function of the attacker model. For instance timing attacks through cache line latencies or other side channels are

not covered if we solely consider the clearance of data but no temporal aspects. An attacker with capabilities outside the attacker model is literally incalculable.

Type systems can be divided into two groups: flow-sensitive and flow-insensitive type systems. Context sensitive type systems are not considered here, they take the context from which a function is called into account, but the analysis is typically flow-insensitive. When we subsequently use the term *flow sensitivity*, we mean *control flow sensitivity*. The next sections describe the two, compares their advantages and disadvantages and gives our reasoning why we have chosen a flow-sensitive type system.

2.1.1 Flow-Insensitive Type System

Flow-insensitive type systems use type inference rules to reason about programs. They are also known as static type systems and are typically used in strongly-typed languages to prove correctness regarding data type compatibility. In a static type system, as the name suggests, the types of variables and expressions are static. They have a fixed type and their type never changes, neither during typing nor later at run time. Consider, for example, a simple type system for integer expressions: If two expressions e_1 and e_2 have type *int* and there are the following inference rules:

$$\frac{\gamma \vdash e_1 : int \quad \gamma \vdash e_2 : int}{\gamma \vdash e_1 + e_2 : int} \qquad \frac{\gamma \vdash e_1 : int, \gamma' \quad \gamma' \vdash e_2 : int, \gamma''}{\gamma \vdash e_1 + e_2 : int, \gamma''}$$

flow-insensitive flow-sensitive

Then, we can deduce that the whole expression is also of type *int*. We say that a program is well-typed regarding γ if all its parts (statements, expressions, variables, etc.) are typeable with respect to γ. We use γ to denote the environment, i.e., the binding of variables and expressions to their types.

Please note that in the flow-insensitive case γ does not change and that the order in which e_1 and e_2 are evaluated does not matter. The flow-sensitive inference rule, on the other hand, forces the evaluation of e_1 before e_2 can be evaluated. Moreover, the environment γ might be changed, depending on e_1's evaluation, resulting in a potentially different γ'. That, in turn, is used to evaluate e_2 afterwards. Evaluating e_2 result in a new environment γ'', which is also the environment after the whole expression. To emphasize this again: With flow-sensitive typing the environment might change depending on the evaluation, with flow-insensitive type systems this is not possible.

A type system can be used to check a broader range of properties, for instance, secure information flow. In this case we instead assign security levels in place of data types to all variables and expressions. We use a finite lattice to represent the security levels (partial ordered set of security levels, least upper bound $\sqcup$, partial order relation $\leq$, with `a` $\leq$ `b` iff information is allowed to flow from `a` to `b`). Variables are typed depending on their clearance, for instance, a variable containing high classified data is typed *high* while another one holding public data is *low* typed. Figure 2.1 shows an example of a well-typed program and a second that is not typeable (assume the name and the type of the variables correspond, i.e. the variable named `medium` has been typed *medium* as well).

```
medium = low;       low = high;          lattice:    high ≤ medium
high = medium;      low = 0;                         medium ≤ low
```

Figure 2.1: A well-typed and an untypeable program

In the first example no information is leaked. Statement 1 assigns a *low* expression to a *medium* typed variable, so the information from variable `low` is copied into variable `medium`. Now *low* classified data is kept in a *medium* classified variable, thus the security level of the copy is raised to *medium*. The whole statement is also typed as *medium*. In the second statement, the same happens again: `medium`'s data is copied into a high classified location, typing the statement as *high* too. As you can see information flows from bottom (*low* variables) to top (*high* variables) are allowed. The opposite is not possible as the second example shows. There the first statement would assign a *high* expression (the variable `high`) to a variable with lower clearance (`low`). This is not typeable ($high \not\leq low$) and therefore the whole program is not well-typed as well.

Nevertheless the second program is secure, no information is leaked: the next statement immediately overwrites the temporarily leaked data. But since types are static, this cannot be expressed in such a type system. The reason is that we need to know in which order statements are executed, which requires a flow-sensitive analysis. This is the major drawback of static type systems: they do not consider this additional information. But with static types it is much easier to type program constructs that flow-sensitive type systems cannot handle easily. The best example are loops: statically easy to type, with flow-sensitive type systems this is really hard to do as it requires loop unrolling or fixed-point calculation to type correctly. Sometimes static type systems yield better (or: at all) results than flow-sensitive ones.

To overcome the problem of the sequence of statements we can transform the second program into static single assignment form (SSA) [HS06]:

```
low_1 = high;
low_2 = 0;
```

Statically typing, together with SSA, can type programs that *temporarily* violate the security policy. Unfortunately this is only true if no pointers are involved. As far as we know, pointers are beyond the means of flow-insensitive type systems (see Section 2.2).

2.1.2 Flow-Sensitive Type Systems

In flow-sensitive type systems, the sequence of statements is taken into account. Each variable is assigned a type or label and, while type inference, these labels might change. Depending on the property of interest, the labels have different meanings.

We are interested in checking the confidentiality and integrity of a program. Warnier et al. describe a flow-sensitive analysis for information-flow security [JPW05]. The concrete value of a variable is no longer considered, instead its security level is used as its label. Then the program is abstractly interpreted, computing security labels instead of concrete values. The following example clarifies this:

```
a = 3;          a {low}
b = 5;          b {high}
c = a + b;      c {high} = {low} ⊔ {high}
```
code *security labels*

The type of variable `c` is H, since it reveals information about `b`, which is of type H. In order to type the statement `c = a + b` the variable `c` must be at least of type H. While the program is abstractly interpreted, it is verified that the security policy is not violated. Consider again the following program, which was not typeable with a flow-insensitive type system:

```
–  –  –  –  low {low} high {high}        lattice:
low = high;                                  ⊤
–  –  –  –  low {high} high {high}          high
low = 0;                                     low
–  –  –  –  low {⊥} high {high}              ⊥
```
code *security labels*

The code is shown left, and the variables and their labels on the right. Before the first statement `low`'s label is *low*, and `high`'s label is *high*. After executing the first statement the label of `low` has changed to *high*, since the variable `high` holds high classified data (its label is *high*) and it has been copied into `low`. With a static type system this assignment is untypeable, since types are static and cannot change, whereas dynamic type systems allow labels to change. The second statement changes the label of variable `low` a second time from *high* to the label of the integer constant `0`. A constant does not reveal any information, it has the lowest possible label, which is typically denoted with $\bot$. Thus `low` is labeled with $\bot$ after the second statement and no information is leaked. This system assumes that an attacker cannot inspect variables during execution; he sees only the final outcome of a program. Otherwise he could observe the value of `high` by reading `low` immediately after the first statement.

The great advantage of flow-sensitive type systems is the ability to allow temporary leakage of information, as long as this leakage is safely overwritten later on and before information is exposed to the outside. Difficult to handle are control structures like loops, break and continue, since they require to selectively skip statements. Pointers are another challenge, as described in the next section.

The primary goal of our approach is to verify kernel code. System calls are known to terminate early, therefore long running loops should not be used. We do expect an extensive use of pointers. Due to its ability to analyze pointers, we have chosen a flow-sensitive type system for our analysis of L4.

2.2 Pointers

Pointers are a major challenge to static analysis. Well-known as *alias analysis*, it has been shown that precise may-alias analysis is NP-hard [Hor97]. Two variables may be alias, i.e., may share the same memory location. When adding dynamic memory allocation it becomes undecidable in general.

As explained in the previous section, we abstractly interpret the program and abstract from all concrete values. We replace them instead with their security level. However, this is not possible with pointers. They point to a variable, and

the information where they point to is strongly needed. Assume the following program:

```
if( high ) {
    p = &a;
} else {
    p = &b;
}
```

After this statement we do not know whether `p` points to `a` or to `b`. `*P` and `a` may be aliased, or `*p` and `b`. In order to precisely analyze data flow we have to know where `p` points to. When writing to `*p` we want to know whether `a` or `b` is changed. Therefore we reintroduce part of the concrete values to increase precision. When variables might be aliased, it is possible to access the same memory location in different ways, which makes information-flow detection much harder.

2.3 JFlow and Jif

Closely related to tool support for secure information flow in C++ code is JFlow/Jif [Mye99], which deal with Java. A significant part of Java is supported and extended by adding source code annotations to variables, expressions, objects, etc. This newly defined language, Jif, is processed by the JFlow compiler, which does source-to-source translation from Jif to Java. Most of the annotations are checked and removed at compile time. In some situations, in particular when labels cannot be inferred statically, runtime checks are added.

Although labels can be inferred automatically, there are still many annotations required to clearly state the policies regarding who may read which data. No doubt, policies have to be declared, but probably not so intrusive. From our point of view adding too many annotations and labels is cumbersome for most programmers and therefore not used by many of them. It also complicates the readability of code, thereby increasing the chance of adding errors through these hard-to-read annotations. We tried to separate the analysis of information flow from the design and verification of a concrete information-flow policy.

2.4 CQual and CQual++

More similar to our work than JFlow is the collection of static analysis tools that include CQual[FTA02] and Oink [DSWM]. CQual is a tool developed by Jeff Foster et al., which lets programmers add user defined type qualifiers like `$tainted`, `$untained` or `$notnull` to a C program. This enrichment of the C type system enables CQual to infer better type information and check if the annotations are consistent. Oink, and its main tool CQual++, is a data flow analysis framework for C++ similar to and inspired by CQual. Both are very powerful, since they cover C++ and C to an impressive extent. Some of CQual++ key features are:

- **Polymorphic data flow**: Multiple calls to the same function do not interfere, i.e., each call results in a separate flow graph.

- **Instance sensitivity**: Objects of the same type are not folded into one, but treated individually, thus assigning to a member of one object does not influence the other object.
- **Not flow-sensitive**: Oink is not flow-sensitive, so the order in which statements occur is ignored. Thus type qualifiers, like `$locked/$unlocked`, that require some state information, are not supported by Oink. CQual allows state-aware qualifiers, but only to a limited degree (recall: CQual accepts only C, not C++). The Oink authors point out that static single assignment form (SSA) solves the problem of flow sensitivity for stack variables, but not for heap allocated variables.
- **Laundering**, i.e., passing data flow through the control flow, is not detected: Oink does not consider data to ever flow into the program counter. The same is true if data flows through array lookups, e.g., an array used to compute the identity function (the data does not flow through *the elements* in the array, but through the *position* of an element).

CQual and CQual++ are more powerful than our tool regarding supported language features of C and C++. However, CQual analyzes only C, and CQual++ is flow-insensitive. Although CQual is flow-sensitive the analysis of pointer aliases itself is flow-insensitive. It is intuitively clear that the more precise (thus the more difficult) the alias analysis is, the better the data flow can be tracked. In our tool we use a lot of the ideas found in CQual *and* CQual++.

2.5 CCured

CCured [Wei03, NCH+05] makes C safer by adding more type information to pointers in order to avoid memory errors. Most of the checks are performed statically at compile time. In case this is not possible, runtime checks are injected. Pointers are automatically classified according to their usage. There are three types:

SAFE pointers are either 0 or point to a valid object. They are similar to Java references;

SEQ pointers are used to access arrays and to perform pointer arithmetic. Non-null and boundary checks are required at runtime; and,

WILD pointers are those that do not provably belong to one of the others (e.g. instead of pointer arithmetic integer arithmetic is used).

In [NMW] the authors show that over 80% of all pointer operations are **safe** and less than 1% belong to the **wild** category. This clearly shows that most C programs *are* pointer safe, but it cannot be *proven* that they are. Since our tool is not capable of supporting pointer arithmetic up to now, the approach shown in the CCured project is not yet usable for us. We intend to integrate their promising ideas into our tool to trace information flow correctly in the presence of pointer arithmetic.

Chapter 3

Design

Our goal is to verify the absence of information flows that violate a given information-flow policy. Information, or data, is kept in variables and is classified according to some security policy. We want to know where data is copied to and that no confidential information is leaked. Our approach, in short, is to abstractly interpret a program and simulate its behavior, especially the operations performed on its data. While doing so, we constantly keep track of how data spreads through the system. In general we try to answer the question: *Where does data come from and where does it go?*

This thesis bases on the research by Warnier et al.[JPW05]. We have also integrated the idea of an universal lattice [HS06]. From our point of view, there are at least two possibilities how to label variables:

1. When we know in advance which security levels exist and how the variables are labeled according to these levels, we can abstractly interpret the program. Afterwards we are able to verify that the final labels of the variables are at least as high as their initial labels (defined by the security policy). Otherwise we found a potential information leak.

2. Use the most general labelling - each variable is represented by an unique identifier (e.g. its name, including the fully qualified scope). Then the label of a given variable is a set of these identifiers that contributed to the value of a variable. In this case, the lattice is the power set of all program variables, with subset ($\subseteq$) as $\leq$ and union ($\cup$) as $\sqcup$.

The advantage of the second approach is that it provides a comprehensive analysis of data flow through the program. At the end we are able to separate the deriving of information flows (abstractly interpreting a program) from the check whether the information-flow policy holds. This is closely related to *data flow analysis*: if someone is able to perfectly calculate the data flow of a program, then information leakage detection is merely declaring a policy stating which flows are illegal and comparing the policy with the data flows we found.

Our modification to the approach of dynamic labeling is to replace the security labels (*low*, *high*, etc.) with variable identifiers corresponding to the variables that influenced the information stored in the variable in question. Figure 3.1 shows line by line the difference between security labels and data dependencies. Assume that the variables `guard`, `foo` and `bar` are already initialized and that `guard` is labeled with *high* and `foo` and `bar` are labeled with

`int i;`	`i` $\{\bot\}$	`i` $\{\}$
`if( guard ) {`	`guard` $\{H\}$	
`  i = foo;`	`i` $\{L\}$	`i` $\{foo\}$
`} else {`		
`  i = bar;`	`i` $\{L\}$	`i` $\{bar\}$
`}`	`i` $\{H\}$	`i` $\{foo, bar, guard\}$

Figure 3.1: A program with security labels and with data flow information

low. The left part of Figure 3.1 displays the source code, the middle part depicts the label of the variable under consideration and on the right side the data dependencies are shown. Without going much into detail, the following happens: Variable `i` will be labeled with *low* in the *then*- and *else*-branches, but due to the *high* label of `guard`, the label of i will be raised to *high* after finishing the `if`∼`then`∼`else` statement, denoted by the final `i` $\{H\}$ at the bottom of the middle column.

In the case where data flow information is added, `i` will be labeled with foo and with *bar* in each branch. Then they are merged and the information flow from the guard will be added, so that after the if statement `i` is labeled with $\{foo, bar, guard\}$ denoting that information from all three variables contributed to the value of `i`. If we map foo and *bar* to L, *guard* to H and take the least upper bound of this set, we get the same result: `i` is labeled H.

We have to merge both branches and we must add the information of the guard, since we do not know which branch will be taken at run time. Therefore we have to assume that both branches might be taken and that both of them might change data dependencies. A more detailed discussion is given in Section 4.3.

Conventional data flow analysis concentrates on the concrete values of expressions when used in connection with optimization techniques like constant propagation/folding and dead-code elimination. In our case we abstract from the exact value of a variable because we do not need to know the value, but only it origin; instead of being interested in the information stored in a specific variable, we want to know where this information *came* from. So we use abstract values describing which information is stored in a variable by using a set of all variable IDs that contributed to the value of a specific variable.

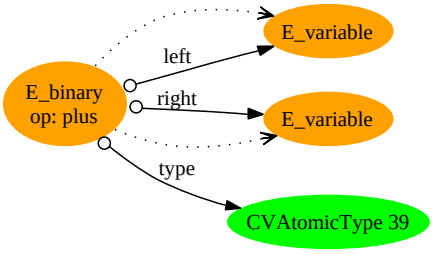
We use the tool chain cpp, Elkhound/Elsa and Olmar to create an abstract syntax tree of the program to analyze. The single steps are the following:

1. Cpp preprocesses the C++ source code, since Elsa cannot do that.

2. The Elkhound/Elsa parser generator is used to construct an abstract syntax tree (AST) from the preprocessed source code.

3. Olmar makes the AST accessible to Ocaml programs as an Ocaml data structure. Alternatively we could have implemented our algorithm in C++ and traversed the AST directly. We used Ocaml for the reasons given in Section 4.1.

Our approach is to analyze a program at the level of its abstract syntax tree (AST). Therefore we implemented a *visitor* pattern that traverses the AST from top to bottom, gathering information about data dependencies. The tool chain



AST representation of a summation (without the dashed arrows). The visitor sees the `E_binary` node, proceeds with the two child nodes and adds the two dashed links. Unrelated information, like type information, are ignored. Later only the dashed links are used.

Figure 3.2: Summation of two variables as seen in the AST.

that we use contains an example AST iterator which traverses the whole tree visiting every node exactly once. We used this iterator as our starting point.

Our visitor traverses the AST of the program from top to bottom and extracts data dependencies (see Figure 3.2, dashed links). The resulting graph is much smaller than the whole AST. Compare the dashed lines in Figure 3.4 (added by the visitor) with the total number of edges. Irrelevant nodes have even been removed from that figure. Still there are many nodes and edges we have to traverse, but which are unrelated to information flow (e.g. statement and fullExpression nodes), since all related information is embedded in the subexpression nodes. In fact, only a very small fraction of all nodes is needed at a time to track information flow.

The reason for this is interesting: Information cannot flow between AST nodes of different statements, because all information is stored in variables. On the contrary, subexpressions carry information to the enclosing expression. The importance of this observation will become clear once we have seen how the *visitor* and the *memory model* interact. Of course information *do* flow between variables when an assignment occurs, but in this case data is copied from one variable to another. Variables in the AST are represented as leaf nodes that refer to our memory model (see Section 3.3). Copying data from one variable to another is therefore made explicit. For now we assume that statement boundaries are information-flow boundaries as well. Therefore we need to know only the nodes of one statement at a time and proceed to the next statement only after we have finished the current one.

Variables represent the state of a program and are kept in a *memory model*. It keeps track of the analyzed program's state and is updated whenever a state change happens. Updates only occur when assignment operations are performed, which is the sole way to change the state of a program (see Section 2.1.2). The memory model is an abstract representation of the memory consisting of abstract storage locations. Each memory cell represents a variable (a detailed description and the reasons why we model it in this way is given in Section 3.3). You can see in Figure 3.3 how the cell of an integer looks like:

location	type	name	label
1	int	i	{}

A location, a type, a name and a label.

Figure 3.3: Integer representation as an abstract storage location.

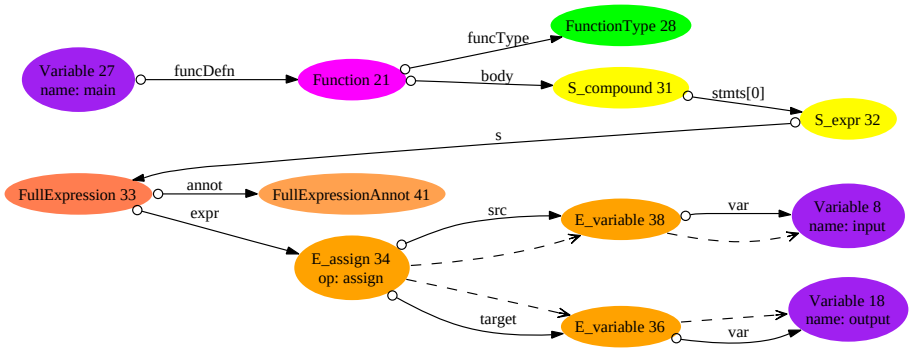


Figure 3.4: Main() function with one assignment statement and the data dependencies caused by that assignment (dashed arrows)

The most important part of an abstract location is of course its label, since it describes which information has already flown into that cell. The label is a set of other locations from which this variable received information. In the given example (see Figure 3.3), the variable `i` contains no information yet and is therefore uninitialized (its label is {}). The next section will present a complete example how the *AST visitor* traverses the AST and of how the *memory model* is involved.

3.1 An Introductory Example

The following example gives a basic understanding of what the AST and the memory model look like:

```
int input;
int output;

void main () {
    output = input;
}
```

Assume `input` is a variable that received some data as program input, maybe from the program's arguments or from a file. `Output` is a variable that represents program output. The single statement `ouput = input;` copies data from `input` to `output`. In this statement, there is an information flow from `input` to `output`.

A simplified AST is given in Figure 3.4. At some point, the visitor arrives at node 27, the main function. It continues with node 21, a function node and then proceeds to the body of the function, a statement compound (node 31). Then, it iterates over all the statements. In our example, this is only one statement (node 32). Until now we have not seen any information flow, because information flows only within statements. The most interesting node is node 34, the actual assignment expression. It has two child nodes, the source and the destination expression.

When the visitor encounters this node, it first continues to evaluate the source expression (node 38) by invoking an expression function with the source

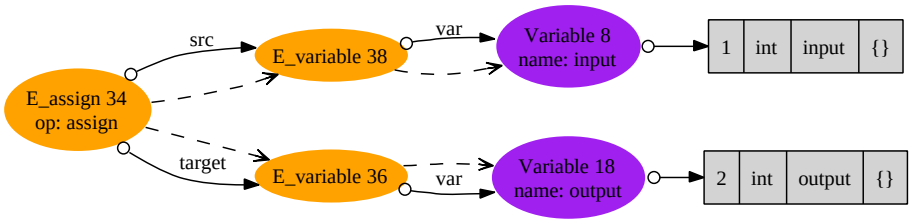


Figure 3.5: The connection between the AST and the abstract memory.

expression as its argument. This in turn evaluates the expression, here a `E_variable` expression, and adds a link between node 38 and node 8, denoting that node 38 depends on node 8. Then, the expression function returns and is called again, but this time with node 36 as argument – the target expression. At the end, a link between node 36 and node 18 is also added.

Normally, the source or target expression expand to a much larger subtree, resulting in a longer run, but the result is the same: Two links, one from the source expression to somewhere and one from the target expression.

Until this point the memory model was not involved, we just added some links to the AST. When the assignment is finally simulated, the state of the program will change and data will be copied from some memory locations addressed by the source expression to some locations addressed by the target expression.

To find out where to read and where to write, we follow the dashed links into the subtree of the source/target expression, going along the edges until we finally reach its leaves – variable nodes. These, and only these, contain the data to copy; all the other AST nodes in between are just used to structure the access to the variables/the memory locations.

Since each variable is bound to an abstract storage location, we can read or write these locations (a detailed description about reading and writing variables is given in Section 4.2). In the example, the source and the target subtree are very small. Later we will see that, when pointers are considered, *both* the source and the destination may produce a set of variables (more precisely: abstract locations). Reading from an abstract storage location means receiving the label of that location, i.e., those locations from which this one has already received data. To write to a location means to replace its label (strong update), because older information is overwritten by newer information. Later we will see that there are also weak updates (see Section 4.3), when it is uncertain if a location is written of not (for instance: `if( boolean ) a = 0;` – `a` may or may not have been written).

location	type	name	label
1	int	input	{}
2	int	output	{}
3	void	fun_main	{}

location	type	name	label
1	int	input	{}
2	int	output	{1}
3	void	fun_main	{}

Figure 3.6: The memory before and after performing the assignment operation.

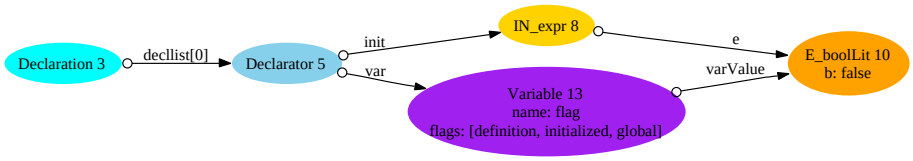


Figure 3.7: AST of: bool flag = false;

3.2 The Visitor

The visitor traverses the abstract syntax tree (AST) from top to bottom invoking an appropriate function for each node. It starts at the root node (a translation unit) and goes down through the functions, statements and expression to variables. The major task of the visitor is to extract data dependencies from the AST, getting a smaller information-flow graph for single statements.

Most of the time, some dependencies between nodes are added. These are used later for evaluating assignments. For each statement are the expressions linked to their subexpressions, down the AST until we reach variables or other leaf nodes, like literals (constants). When between all necessary subexpressions links are added, the statement itself is evaluated. Solely assignments and initializations modify the memory model, the other statements do not.

The easiest example of information flow are constants, like `E_boolLit`, representing a boolean literal. They contains no information per se and therefore no information can flow at all. A link is added between this node and the bottom symbol $\bot$ (in this case, the bottom symbol for booleans, $\bot_b$). When this node is later used, e.g., during initialization of a variable, then the variable will get the label of this node, the bottom symbol. The following example shows this approach:

```
bool flag = false;
```

A simplified AST is given in Figure 3.7. It starts with a declaration node, followed by a declarator. That in turn has two child nodes, the variable to declare and an IN_expr. Finally both link to an `E_boolLit` node. The visitor traverses the AST and does the following:

- Enter this subgraph at node 3, a declaration.
- Iterate over its children (in this case there is only one, node 5).
- Get the type of the declared variable, parse it and allocate memory on the stack. Here we also check if the variable is an input variable (can be read uninitialized) or output variable (visible after program completion)
- If there is an `IN_expr`, process it.
- Continue with the variable node itself. Perform the initialization if there is one. This is almost equal to an assignment, which means that the state changes. Data is copied from the init expression into the newly allocated variable and the memory model is changed.

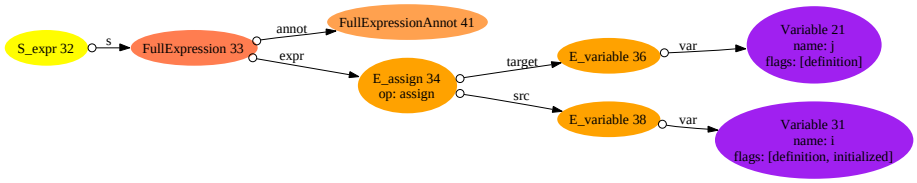


Figure 3.8: AST of: j = i;

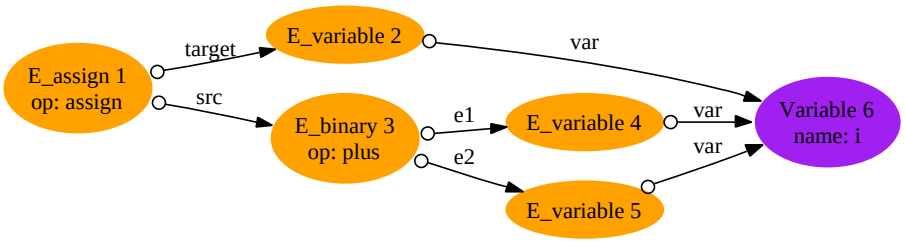


Figure 3.9: AST of: i = i + i;

The example in Figure 3.7 sketches how the visitor traverses an AST and performs operations on the nodes. When state changes are to happen (initializations and assignments), they are reflected by corresponding updates to abstract locations in the memory model. The next example (Figure 3.8) shows the more common way to change the state of program variables: by assignments.

When we have the assignment of an expression to a variable (an r-value to an l-value), then the visitor traverses the AST, encounters this `S_expr` node and proceeds with the fullExpression child node. Later, when returning from the subtree, it adds a link between node 32 and 33. At node 33, the annotation node is evaluated first, since temporary variables are declared through these annotations. We need to process them first before we can use these up-to-now unknown variables during the subsequent evaluation of the expression.

In the example there are no temporary variables, therefore we do nothing at node 41. The next node is node 34, an expression of type `E_assign`, with two children. One is the source expression (the one the data will be copied from) and one is the target expression (where the data will be copy to).

First the source expression is evaluated (node 38), which is an `E_variable` referring to the real variable. This additional indirection is used to distinguish between expressions and variables. In the expression `i = i + i` the variable `i` is used three times, but the corresponding `E_variable` nodes are different. Both operand expressions of the summation and the target expression are different, however, all three point to the very same variable (see Figure 3.9).

Coming back to the example, this means that a link between node 38 (the `E_variable` expression) and node 31 (the "real" variable) will be added. Member variables and static variables are treated specially. Similarly, node 36 and 21 are linked when processing the target expression.

Finally, the assign operation is done by the memory model. It follows the links of the source expression until variables are encountered. Then, their labels are copied to the target variable.

arithmetic types	
integral types	floating point types
signed char, short int, int, long int unsigned char, unsigned short int, unsigned int, unsigned long int char, wchar_t bool	float, double, long double

Figure 3.10: C++ Fundamental types

We have now seen how C++ source code is represented by an abstract syntax tree and how the visitor traverses this tree. The primary goal of the AST visitor is to extract data dependencies between AST nodes and collect them in a smaller dependency graph for single statements. We have also seen that the state of a program changes when assignments are carried out, i.e., the values of variables are changed. Thus we need a memory model that keeps track of whenever an assignment happens. In the next section, we present one such model and give our reasoning for it.

3.3 The Memory Model

Statically checking a C++ program is done at compile time, or more precisely after preprocessing the source code. Depending on the language specification, the target platform and the compiler variables may have different sizes and the memory layout is implementation dependent. This is not problematic for our purpose, we do not have to know where precisely a variable is stored in memory if we know that it is stored and we can address it somehow. Therefore we use abstract storage locations as a layer of indirection.

We model an infinite pool of fresh locations, and when we encounter a new variable declaration we allocate the next fresh location for that variable. We distinguish between heap and stack allocated variables, for instance new objects are allocated on the abstract heap, but the pointers to them are placed on the abstract stack. Therefore local variables are removed once they go out of scope. Memory addresses and memory layout becomes abstract, nevertheless, this can be done very precisely, consider bit fields or casting an int to a char. With a sufficiently accurate memory model, this can be modelled correctly. Note that casts to and from pointers are not covered, because pointers themselves *are* related to the abstract location where they point. If it proves necessary, the abstract locations can be mapped to real memory later on. Sometimes more abstract locations are added than required in reality, as such, they are merged by applying an alias analysis to determine equivalence.

3.3.1 Fundamental Types

C++ defines fundamental types as shown in Figure 3.10 [C]. Variables of fundamental types are mapped to abstract storage locations (ASL) that consist of an ID, a name (not required, but useful for debugging), a type (as listed above) and a list of the other ASLs from which this cell already received information.

```
if( input1 ) {
    tmp = input2;
} else {
    tmp = 0.0;
}
output = tmp;
```

location	type	name	label
1	bool	input1	{}
2	double	input2	{}
3	float	tmp	$\{2,\bot_f\}$
4	float	output	$\{2,\bot_f,1\}$

Figure 3.11: Indirect information flow through if∼then∼else guard

Consider the example given in Figure 3.11. The final label of `output` is $\{2,\bot_f,1\}$, which means it depends on or received information from:

- ASL 1 (`input1`) – indirect flow through the guard,
- ASL 2 (`input2`) – direct information flow through assignment,
- $\bot_f$ – float literal 0.0.

There is also a potential loss of information when copying `input2` (double) into `tmp` (float) if float and double have different sizes. This might either be ignored (which is safe with respect to an information-flow analysis and the security policy) or reported.

A few remarks: When the program runs, it gets some input and produces some output. When it is statically checked, this information is not yet available. Therefore we introduce some variables that are annotated as being input or output parameters by putting *@in* or *@out* as a comment in the source code in the same line as their declaration (See Section 3.4).

3.3.2 Pointers and References

Without pointers no memory model is necessary. By using a transformation to static single assignment form (SSA), information flow can be modelled solely within the abstract syntax tree. With pointers, variables may be aliases. Since we abstractly interpret the program, the concrete values of variables are replaced with their label. But pointers and references are different, in that they do not contain concrete values in the sense above but *point* to some other variable or abstract storage location that in turn holds the information (see Section 2.2). Consequently, their values matter and need to be modelled as well. To reflect their behavior we model them with *two* labels of different kind:

1. Its normal security label, similar to other variables. It is a set of locations that *influenced* where the pointer points
2. Additionally, a set of locations *where* it might point

The first label is similar to that of other types, but the second is special and is treated differently. It is added to model to which locations a pointer might point, i.e. concrete abstract locations. To make this clear again: Pointers have two labels, the first describes which locations influenced it and the second where the pointer points. The first label is only changed by the guards of conditionals, like `if∼then∼else`.

```
int input;       // @in
int output;      // @out
int* p = &output;
*p = input;
```

location	type	name	label
1	int	input	{}
2	int	output	{1}
3	ptr{2}	p	{}

```
int input        // @in;
int output       // @out;
int* p = &input;
output = *p;
```

location	type	name	label
1	int	input	{}
2	int	output	{1}
3	ptr{1}	p	{}

Figure 3.12: Copying input via pointer p to output

Figure 3.12 shows two very similar examples, both copy `input` via a pointer to `output`. In the first case, the dereferencing takes place on the l-value, in the second case on the r-value.

Figure 3.13 shows a pointer that is set conditionally. Thus, it depends on the condition and potentially points to more than one location. At runtime it will point exactly to one location, but at compile time it is unknown to which it will point and both locations have to be considered.

```
int *p;
if( true ) {
    p = &input1;
} else {
    p = &input2;
}
output = *p;
```

location	type	name	label
1	int	input1	{}
2	int	input2	{}
3	int	output	{1,2,$\perp_b$}
4	ptr{1,2}	p	{$\perp_b$}

Figure 3.13: P points to input1 or input2, depending on the if-guard

The variable `p` depends on the if-guard, here a boolean constant with the label $\perp_b$, thus the label of `p` is $\perp_b$. Additionally, after the `if`∼`then`∼`else` it is unclear whether `p` points to `input1` or `input2`, therefore its second label is {1,2} (`p`'s type is ptr{1,2}).

References *are* pointers as well. They can be considered as `const` pointers that never can be `NULL`, but nevertheless they have to be treated like pointers. Figure 3.14 shows how a reference might point to more than one location. They are similar to pointers in the sense that they "point" to another variable (abstract location), but they are never explicitly dereferenced. This happens on the fly when reading or writing a reference. They never exist uninitialized and never refer to another reference. In the example, we use a function that takes two references as arguments and conditionally returns one of them to get a reference pointing to one of the two parameters.

3.3.3 Compound Types – Classes

Compound types like classes and structs comprise several members that are either of a fundamental type or again a compound type. A class might have some base classes from which it inherits. A struct is a class with all its members declared to be public as default. An object variable has a complex structure,

```
int input1; // @in
int input2; // @in

int& foo( int& a, int& b ) {
    if( true ) {
        return a;
    } else {
        return b;
    }
}

int& i = foo( input1, input2 );
```

location	type	name	label
1	int	input1	{}
2	int	input2	{}
5	ref {1}	a	{}
6	ref {2}	b	{}
7	ref {1,2}	i	$\{\bot_b\}$

Figure 3.14: A function conditionally returning a reference

consisting of a couple of abstract storage locations that belong to its members and that are grouped together. The members are addressed relative to this variable. Consider the following class C and the object `c` of this class:

```
class C {
    int i;
    char ch;
} c;
```

location	type	name	label
5	cmp C	c	{}
5 47	int	c.i	{}
5 56	char	c.ch	{}

Figure 3.15: A class with two member variables

Every object of type `C` has the same layout: Two member variables, an integer `i` (AST ID 47) and a character `ch` (AST ID 56). In the example, the object `c` gets the abstract storage location 5. Different instances of class `C` are placed at different memory locations, assume we declared two variables of type `C`, then the memory would look like this:

```
class C {
    int i;
    char ch;
} c1,c2;
```

location	type	name	label
5	cmp C	c1	{}
5.47	int	c1.i	{}
5.56	char	c1.ch	{}
7	cmp C	c2	{}
7.47	int	c2.i	{}
7.56	char	c2.ch	{}

Figure 3.16: Two objects of type C

One object is placed at location 5, the other at location 7. But the member variable `i` is put at location [5.47] if it belongs to object `c1` and at location [7.47] if it belongs to `c2`. This so-called *instance sensitive* model allows us a fine grained control where information is copied too. An alternative way would be to treat all objects of the same type as only one object, folding all instances into one. This would speed up things a lot and simplify the memory model too, but at the price of a less precise information-flow analysis.

One may ask why there are at all entries of type `cmp`, since the sum of all members already holds all information stored in the whole object. This is true

```
class A {
    int i;
} a;

class B1 {
    A a;
} b1;

class B2 : A {
} b2;
```

location	type	name	label
1	cmp A	a	{}
1.13	int	a.i	{}
3	cmp B1	b1	{}
3.155	cmp A	b1.a	{}
3.155.13	int	b1.a.i	{}
5	cmp B2	b2	{}
5.22	cmp A	b2.a	{}
5.13	int	b2.i	{}

Figure 3.17: Base class A, aggregated by B1 and inherited by B2

and normally these artificial entries are not required. We exploit these entries to model the `this` pointer correctly. When we are within an object `o` that itself is a member of an enclosing object `p`, we can only use the `this` pointer of `o`. All the data dependencies we see and all the information flow we analyze is *relative* to `o`. Later when we have left `o` and return to the outer object `p`, then the dependencies will become relative to `p`. But within `o` we do not know `p` and therefore need the artificial `cmp` entry in our memory model.

Classes that have other classes as member or classes that inherit are structured as shown in Figure 3.17. Object `b1`, for instance, has a compound member `a` of type `A`. As the memory layout on the right side shows, `a` is embedded into `b1` as a regular member variable. In case of inheritance, the members of the base class are added to the derived class.

Previously, we have seen how the AST visitor traverses the abstract syntax tree and now we know how the memory model looks like on which the visitor operates. Variables are allocated at abstract storage locations, fundamental types are treated straightforwardly and pointers and compound objects are allocated as described – pointers have two labels (where they point and on which locations they depend). Objects have more complex abstract storage locations (like in [3.155.13]) depending on their members. An open issue and subject of the next section is the question where to start with the abstract interpretation.

3.4 Function Invocation

With a memory model and an AST visitor in place, we finally have to start traversing the AST, i.e., to start interpreting the given source code. Since the program is neither run nor do functions get called, how do we know where to start? A good point to start are function entries, since they are good boundaries. When a function is called it gets some arguments passed to operate on. Beside side effects the only information that flows to and from a function is through its arguments and the value it returns. We annotate a single function with an *@entry* tag denoting this function as the point where the abstract interpretation should start.

Additionally, since the function is not really running, but is taken from its environment and abstractly interpreted, some environment is needed. Usually, a couple of surrounding data structures are missing or not properly set up, so some adaption has to be made by hand. No “real” input exists that could be

passed to the function, thus some variables are annotated with *@in* to denote that these are input variables which will hold input data later at run time. Variables receiving output are annotated with *@out*, indicating that after program termination these variables will be visible to the outside.

This leads directly to the question how to model function calls. We found two ways, which we refer to as *early* and *late evaluation*, both have their advantages and drawbacks. Using *early evaluation* means we calculate the data flow graph before a function is called. Whenever it is called later, we just put this graph in place to find out how information flows. *Late evaluation* does the contrary: We initially ignore function declarations and when we encounter a function call, we bind the arguments to the parameter of the function and call it. Subsequent calls to the same function will again cause its evaluation, each time with a different set of arguments.

3.4.1 Early Evaluation

Functions are analyzed at the time they are declared. We process the source code in the order it is written. The parameters of a function are treated like input parameters to the program. As concrete arguments are missing at that point in time, the data flow graph will be under-specified. The information flow from the input parameters to the output parameters is calculated and stored at the function node itself. We can view functions as black boxes once we have analyzed them. Arguments are passed in and the output somehow depends on them, but later on we simply put the information-flow graph in place where the function is called. Figure 3.18 shows a stylized function with three in and two out parameters. The data dependency graph of the function is shown by the arrows: Information flows from the first and the second in-parameter to one out-parameter and the second out-parameter depends on the third in-parameter.

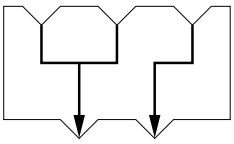


Figure 3.18: A function with three in and two out parameters

When a function calls a second function, we have already evaluated that function before, since we can only call previously declared functions. So we can use the calculated information flows of the second function, replace the parameters by the current arguments, and put the data flow graph in place like a black box. We analyze the given source code linearly, starting at the top and going downwards by traversing the AST, processing function by function and, once we finish, we have a complete information-flow model.

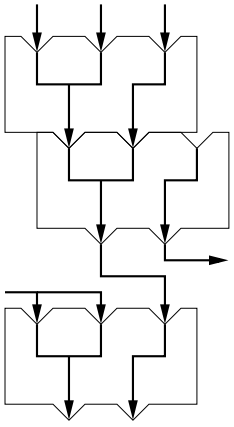


Figure 3.19: Stacking functions together

This approach requires *pure functions* in the first place. If non-local variables are affected, then this would mean that some information flows from or to the outside bypassing the in/out parameters. We can capture this by checking that every read/write operation only affects local variables. When non-local abstract storage locations are read/written, we put these locations in a set of *effect variables* to describe the side effects of a function. More problems arise when pointers are involved. We would have to model all

read/write accesses where pointers are involved, which increases the potentially affected locations drastically

Assume the `memcpy()` function. It copies n bytes from one memory area to another. Its three parameters are the number of how many bytes to copy (n), a pointer to the first byte to copy (src) and a second pointer where the data should be written (dest). When we would have to early evaluate this function, we would not know how much to copy, where to read and where to write. Maybe we can model it, but it would be a rather complex description of all the side effects this function causes and the result would be a abstract description of what a function actually does. This, in turn, is very close to late evaluation, therefore we have chosen the second approach.

3.4.2 Late Evaluation

Late evaluation is more like abstract interpretation. We skip all function declarations until we get to the function annotated with *@entry*. This is typically, but not necessarily, the `main()` function. As mentioned earlier we normally have to supply some environment to the function we want to analyze, so we prepare this environment (data structures, etc.) in the main function before calling the function of interest. When a function is called, we analyze its parameters, allocate them on the stack similar to a call frame and initialize them according to the current arguments. Then, we allocate a return structure for *this function call* and finally invoke the function. In contrast to the former approach (first analyze the function and afterwards put the arguments in place) we call a function with the parameters bound to the concrete arguments. Therefore we can perform all the side effects the function generates immediately and do not have to record them and apply them later when concrete arguments are available. Evaluating functions this way is easier, maybe more precise, but comes at the price that the analysis is no longer linear. We must jump forth and back as functions are called.

When comparing both approaches, the *early evaluation* is more elegant and most suitable for pure functions and functions with minor side effects. Nevertheless, we decided to use *late evaluation*, because the more side effects we get, the more complex their description became. This was especially true with pointers. Additionally, invoking a function every time it is called after having bound the parameters to the current arguments is more similar to abstractly interpreting a program than statically analyzing a function once and reuse the result on every function call.

3.5 Limitations

So far we have explained what we would like to achieve and how we approached it. This section points out the current limitations of our tool and outlines our ideas to overcome them.

Loops are hard to deal with, especially in flow-sensitive type systems, as it requires loop unrolling or fixed-point calculation to type them correctly. Bounded loops, like `for`-loops, are easy to unroll, but unbounded loops, like `while`, are not. Fixed-point calculations are one way, i.e., we gather information about

which side effects are caused by a loop and when they no longer change, we can continue taking this fixed-point as the side effect of the loop.

The current version supports *pointers* in a restricted fashion, only Java-style pointers do work. Neither pointer arithmetic, nor casts from and to pointers are allowed. This also includes arrays, since they are pure pointer arithmetic (`a[i]` is `a + i * sizeof(base type)`). Fixed-size arrays might be replaced with compound objects and iterating through them results in many single statements. Otherwise code can be added to do boundary checking at run time.

Until now we are not able to *skip statements*. Being able to selectively skip statements would help us to support `break`, `continue` and `goto` statements. It would also improve the accuracy of analyzing statement sequences like the following: `return 0; do_something_evil()`. Up to now we are not able to recognize that the second statement is unreachable and that nothing evil happens.

Templates are not automatically replaced by our tool, we had to instantiate them by hand. We did not implement the automatic replacement of templates, because in our case study only two templates were used. Modern compilers do instantiate templates, so if it becomes necessary we will either integrate their solutions into our tool or find a way to factor this problem out and let it be solved by compilers.

Casts, besides pointer-to-pointer casts, are currently not supported. C++ defines some implicit casts, like int to float or int to bool, these we would like to automatically support. The explicit casts in C++ need some more sophisticated techniques, but maybe we can note which type some data had *before* it was casted and use this information later on.

Exceptions do not work and we are not planning to support them in the near future. Since we target C++ kernel code, where no exceptions are used, they are of no immediate concern.

The last point is *assembly code*. Since we focus on static analysis of C++ code, we have not tried to deal with inline assembly. However, with an appropriate model it should be possible to infer date flows through a reasonable part of inline assembly code as well.

3.6 Summary

In this chapter, we presented the design of our tool to statically check information flow. We started with the *AST visitor*, which traverses the abstract syntax tree gathering information about data dependencies. The result is a much smaller graph that we use to update the *memory model* when assignment operation changes its state. We have seen how fundamental data types as well as pointers and objects are presented in the memory. Finally we have had a look at function calls. We have also seen where the current limitations are and how we plan to overcome them.

The next chapter gives an in depth discussion of some interesting implementation details, like how abstract storage locations are read and written and how conditional branches are modelled using a filter to delay memory write operations.

Chapter 4

Implementation

In this chapter we will give a brief justification why we use Ocaml followed by some particularly interesting implementation details. We first explain how memory accesses are modelled and how we read/write abstract memory locations. Then we show how `if~then~else` precisely works. The last part of this chapter gives a detailed list of all the operations the visitor performs on the AST nodes.

4.1 Ocaml

In an early stage of this thesis the question arose which language we should use. The alternatives were C/C++ and Ocaml, for the following reasons. We wanted to *use* an AST, rather than generate it beforehand, so we were looking for some tools that give us an AST in a comfortable way and chose the Elkhound/Elsa framework. Elsa is written in C++ and there is an integrated AST iterator also written in C++. Alternatively Olmar, an extension to Elsa, transforms the Elsa AST into an Ocaml variant, together with an AST iterator written in Ocaml. So we had the choice to use either the Elsa AST iterator, or the one from Olmar, or to write a new one from scratch, maybe in Java. Performance was not a critical aspect, thus all three (C++, Ocaml, Java) would have been fine.

Dealing with ASTs often involves heavy pattern matching, and in this regard functional languages are unbeaten. A variety of data structures are elegantly handled in Ocaml, like tuples, lists and trees. Applying a function to a list or a tree via `map` or `fold` is very convenient. Type safety is also a nice aspect of the functional programming paradigm.

As a remark: Ocaml is not pure functional like, e.g., Haskell. Variables themselves are immutable, but records may contain mutable fields. In this way, state exists and one can assign some value to such a mutable field. In Ocaml `let x = 0` binds x to the value zero, x itself is immutable until newly bound. But `x := 0` assigns zero to the mutable content of x, thereby changing its state.

In the next sections, we will exploit some details about reading/writing to abstract locations and how conditional branches like `if~then~else` are implemented

4.2 Reading/Writing Abstract Storage Locations

Before we can read or write to a location, we have to find it. When we want to access a variable, we use the ID of its AST node and map it to the corresponding abstract location. Member variables are treated equally, the ID we use as the mapping key just consists of the object ID and the member ID (e.g., an object with ID 5 having a member variable with ID 22 would mean that this member of this object is at location [5.22]). After we have found a location, we remove potential reference parts from it (pointers are explicitly dereferenced, references are not). Thus we test if any prefix of the location is known as a reference and if so we replace it by the destination location of this reference. Finally we get a reference-free abstract storage location that we can access. We implemented the following four functions to interact with the memory model:

Peek is the most trivial function, given an abstract storage location, it returns the content, i.e., the type, the name and the label of that cell or it fails if the given location does not exist. No implicit dereferencing is done. Sometimes we need to peek into a memory location when we want to read a reference itself instead of its destination or if we want to "read" uninitialized data to get its type.

Read combines peek with dereferencing. It further checks when we read multiple locations that all their types are equal and combines the labels of multiple locations. Furthermore, we can check if we read an uninitialized variable (its label is {}). When reading such a location we test if this variable is in the set of input variables. If so, the label the read operation returns is the variable itself instead of its label, because read depends on this variable rather than on its data (which is not available at compile time).

Alloc parses the type of the variable/object to allocate, returning the memory layout required for this type. If there is no mapping yet for this variable ID, a new one is created and a new location is allocated. If there is already a valid mapping, it is reused after we have blanked it. This happens every time a function is called: The parameters and local variables are either allocated on the stack or blanked if they are already there. A special allocation is done for the new operator: The object is allocated on the heap, while the pointer to this object is placed on the stack.

Write performs a weak or strong update on abstract storage locations. First we resolve references as explained earlier, then we check if it turned out that this write operation has multiple destinations and if so, we do a weak update, although a strong update might have been requested. Next, we read the location to get its type, because we keep the type of memory location. Consider the following code:

```
if( true ) i = 1; else i = 2;
```

When we merge the branches after the `if~then~else` statement as explained in the next section, we will do a strong update on `i`'s location to write the integer literals followed by a weak update to write the dependency on the boolean literal. This second write would change the type of

`i`'s abstract location from `int` to `bool`, something we do not want. Thus we weakly write the *new* label, but preserve the *old* type, the type we have just read.

Pointers need some special attention, since they have a label and a list of locations to which they point. From the example above we learned that it happens that we have to write an integer location with a boolean to express that the integer location depends on a boolean. All combinations of mixed types are possible. Regarding pointers, we have two interesting mixes (The other two combinations (pointer–pointer and non-pointer–non-pointer) are less interesting):

1. Writing a location that is a pointer with something that is not one.
2. Writing a non-pointer location with a pointer.

In the first case, we add the new label to the label of the pointer and *keep* the list of locations the pointer points to. The second case is the contrary: We take the label of the pointer, apply it to the label of the location we are writing and *drop* the list of locations where the pointer points to.

Writing an abstract storage location is somehow complicated, because we need to consider both the type of the location we are writing and the type of the data that we are going to write.

With these four operations, we manipulate the abstract memory model. Other interesting aspects not described in this thesis are how precisely we find the abstract location given an AST ID, how we handle functions that return references and how dereferencing works.

4.3 Conditional Branches

When we statically type check `if`∼`then`∼`else` (other conditional branches like switch statements are similar), both branches (either the then or to the else path) have to be analyzed independently from each other but within the context of the guard condition that decided the branch. We cannot simply process first the then path and when we are done continue with the else path, because the side effects of the then branch would influence the else branch. Later at run time *either* the then *or* the else path will be taken, but not both. The two paths are mutually exclusive, therefore they need to be evaluate independently.

We use the very same initial state of the memory model after the evaluation of the condition and evaluate both branches in parallel. We achieve this by adding a layer on top of our memory model that acts like a filter: Read operations are propagated down to the layer beneath, but write operations are captured by the layer and are not propagated immediately. This means that reads come from below, but writes end up in the filter for a while. The following code conditionally modifies an array, see Figure 4.1 to get an idea of how the layer operates.

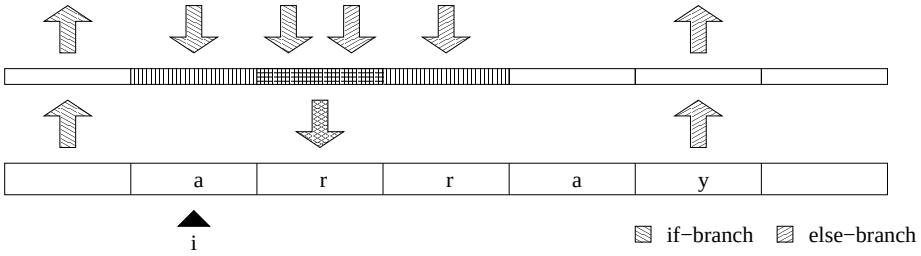


Figure 4.1: Memory access through a filter to capture write accesses

```
if( flag ) {
    array[i] = array[i-1];
    array[i+1] = what;
} else {
    array[i+2] = array[i+4];
    array[i+1] = ever;
}
```

We have to evaluate both paths within the context of the guard, thus the guarding expression is evaluated first. But before proceeding to the then path, we add a new, empty layer on top of the memory. While evaluating the statement compound some write operations will occur. We capture these by the filter and do not let them reach the abstract memory. When we have finished the then path, we remove the layer with all those pending writes and keep it. Now we again add a fresh new and empty layer and do the same with the else path. Afterwards, we remove the else-layer too and end up with an unchanged memory state and two filters, one containing pending write operations from the then branch, the other the pending writes of the else branch. The last step is to merge the two filters and write the result to memory.

This is where *weak* and *strong* updates occur. Strong updates are write operations to an abstract memory location that overwrites the older label by replacing it with a new one. Weak updates are write operation that *may* affect a memory location, but not necessarily. In the example above the array elements `array[i]` and `array[i+2]` may be written, depending on the branch that is taken. They are *never both* overwritten, thus we cannot be sure which of the two locations has changed. This means that they are weakly updated, in the sense that information may be written to the memory location. The new data is added, but does not completely replace the information that is already stored there.

Another example is a variable which is written in the then-branch, but not in the else-branch. After the `if~then~else` statement, it may or may not have been updated. So weakly updating a location means to use the least upper bound of the old and the new label, or in our case: Adding to the already existing set of locations the new locations. If, for example, a variable `i` depends on locations 1 and 3, then its label is {1,3}. If this variable is weakly written by a dependency on location 2, then both labels ({1,3} and {2}) are merged into the new label {1,2,3}.

Merging the two filters from the then and the else branch is now easy to do: If an abstract storage location has been written in only one of the two filters,

we do a weak update on the corresponding memory location. If, however, both filters contain a pending write, i.e., both branches wrote to the same location, we merge these two write operation into one and do a strong update (overwrite) on the corresponding location. See again Figure 4.1: The array at index `i` and `i+2` is written weakly, index `i+1` gets a strong update, because both branches wrote to that location.

So far we have seen the implementation of the operations on the abstract memory, peek, read, alloc and write. The second implementation aspect we presented were the implementation of the `if`∼`then`∼`else` statement. We used layers to filter write operation and delay them until both branches completed, merging the pending writes together. The following section gives a detailed list of what the AST visitor does on the individual AST nodes.

4.4 Visitor Implementation Details

The following part describes in detail the operations on each node. The AST visitor starts with a translation unit node (root node covering a whole source file) and iterates over its children.

translation unit Iterate over all children (top level forms) and collect function nodes. Filter out all functions without a *@entry* annotation. Before starting abstract interpretation, we need to take care of static variables, because they exists *between* function calls and cannot be placed on the stack. In this case we iterate over the whole AST as a list of *all* nodes and allocate static variables on the stack *before* starting to interpret functions. Now we can safely proceed with the evaluating of the function annotated with the *@entry* tag by allocating a return entry, calling `func_fun` with the function node as argument and removing the return entry afterwards.

toplevel form It is either a function definition or a variable declaration or something else (template, assembly, namespace, etc.), everything but the first will fail-fast, since the other constructs are not supported, yet. If it is a function definition, then we use its source code location to find out whether there is an *@entry* annotation. If so, we add this topform's function child node to the list of function nodes that are entry points.

If the toplevel form is a declaration, then it is a global declaration, i.e., variables that are globally visible and that might be input or output parameters. The type of the declared variable is parsed and memory on the stack is reserved. For more details about parsing a type and allocating a variable see Section 3.3.

function function Interpret the function's body – a statement compound. Additionally a function might return an object rather than a mere fundamental value. In this case the object depends on the return statements of this function.

statement function Information flow only happens within a statement, there is never any flow *between* statements. Basically, we create a new empty data dependency graph before processing the statement and remove it afterwards when we come back. The second more subtle thing we need

```
bool flag;  // @in
int i,j;    // @in
int k;      // @out
int* p;
if( flag ) {
    p = &i;
} else {
    p = &j;
}
k = *p;
```

location	type	name	label
1	bool	flag	{}
2	int	i	{}
3	int	j	{}
4	int	k	{}
5	ptr {}	ptr	{}
1	bool	flag	{}
2	int	i	{}
3	int	j	{}
4	int	k	{1,2,3}
5	ptr {2,3}	ptr	{1}

Figure 4.2: Conditionally set a pointer and dereference it. The right side shows the memory state before and afterwards.

to consider are pointers and references: When dereferencing a pointer we get a list of abstract storage locations where this pointer might point. Information can flow from/to these locations. *But* additionally the pointer might have been set conditionally, thus might depend on other locations as well (see Figure 4.2). These indirect dependencies have to be taken into account too. When dereferencing a pointer to read a location we have to remember these indirect dependencies, thus we collect them into a buffer. When we later write a location through a pointer we add the buffered dependencies. Between statements we have to clear this buffer, since dereference dependencies of one statement must not influence the next statement.

The list of individual statements can be found in Section 4.4.1.

fullExpression function First the annotation is processed (via fullExpressionAnnotation function) and then the optional expression function. Finally, we add a link to the expression node, if there is one.

argExpression function This one is called to handle arguments to functions or constructors. We simply invoke the expression function with its child node and add a link between them.

expression function Expressions are the very basic elements of the AST. Some are straight forward to process, others like constructor calls, are very complex. The detailed list is given in Section 4.4.2.

4.4.1 Statement Functions

S_expr A statement expression node has only one interesting child node, a full expression. We evaluate the full expression and add a link to it.

S_skip Simply do nothing.

S_compound This node represents a list of statements, such as function bodies or basic blocks. We iterate over the whole list, calling the same function (statement function) with the child nodes as arguments one by one.

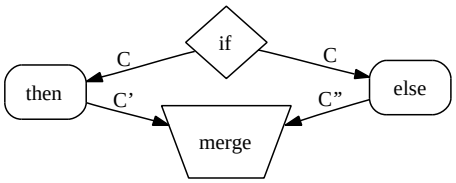


Figure 4.3: Combining the two environments C' and C''

S_if Control flow can take two alternative paths here. The same is true for the ternary operator `?:`, but that one can simply be rewritten as `if`∼`then`∼`else`. Conditional branches are the only situation where pointers or references can get conditional dependencies.

Since we do not know which path will be taken later at run time, we have to consider both paths independently and combine the results. First, the condition is processed by a condition function. Then, we evaluate the body of the if path within the context $\mathcal{C}$, resulting in a new context $\mathcal{C}'$. Proceeding with the body of the else path with the very same context $\mathcal{C}$ we get a new $\mathcal{C}''$ and then we merge $\mathcal{C}'$ and $\mathcal{C}''$ to get the final context. This was explained in greater detail in Section 4.3.

S_return This node comes in two flavors, either its child is a full expression or a statement, depending on whether the function returns a value or an object is returned. In the first case, we evaluate the full expression and add a link between that expression and the function that is going to return here. If an object is returned, then we evaluate the statement, which is a constructor call to the object that will be returned.

S_decl This AST node points to a *declaration* that in turn points to a list of *declarators* that declare new variables. Functions are also declared through variable nodes. Nested functions do not exist and top level declarations are already processed earlier. Static variables have already been allocated, so we skip them here. What remains are local variables: We parse their type, allocate them on the stack and initialize them if we have to. If there is a ctor statement, we invoke this too.

Not implemented are the following statement nodes: S_label, S_case, S_default, S_switch, S_while, S_doWhile, S_for, S_break, S_continue, S_goto, S_try, S_asm, S_namespaceDecl, S_function, S_computedGoto.

Loops are difficult to check statically, since the number of iterations might be unknown. Switch statements can be replaced by `if`∼`then`∼`else`. Break, continue, goto and labels are not supported, they would require to selectively skip statements. An in-deep discussion on these topics is given in Section 3.5. Exceptions and assembly is beyond the scope of this thesis. The remaining 3 node types (name spaces, S_function and S_computedGoto) did not occur so far and are postponed until they are needed.

4.4.2 Expression Functions

E_boolLit, E_intLit, E_floatLit, E_charLit Literals do not contain any information, so they are represented by the bottom symbol $\perp$. It does not matter whether all literals are modelled with $\perp$ or if we have different symbols for different types, e.g., integer literal get $\perp_i$ and floats get $\perp_f$. The advantage of different symbols are preciser type information in memory. In this work we distinguish between bool, char, int and float, but treat long, short, signed and unsigned as one type. This can be further refined when the memory model gets more precise by adding size information to the individual types.

E_this Whenever a member variable or function is used, `this` is implicitly added (or the programmer explicitly did it already). `E_this` has as a child node, the variable that represents the object. We determine the abstract memory location of that variable, put an artificial pointer on the stack pointing to it and add a link from `this` to that pointer. Later when dereferencing `this`, the pointer will be used and correctly resolved.

E_variable An expression denoting a variable, either of a fundamental type or of a class type. If the variable it links to is an enum, we add a special dependency to $\perp_i$, an integer literal. If it is a member and not declared static, we add `this` before it to denote it will be a member access and link to the abstract storage location where this member is placed. Otherwise, we simply link to the variable that in turn will point to its own storage location. In order to know which is the current `this` pointer, we maintain a stack of pointers with the current one on top of it.

E_funCall Denotes a function call. First we get the function to call and its parameters and allocate the parameters on the stack. Then we take its argument list and split the parameter list according to the number of arguments. C++ allows default parameters, so if the number of arguments is less than the required number of parameters, the remaining parameters must have default values. Therefore the first n parameters are initialized with the values of the arguments, and the others with their default values. If a default parameter is an object, we have to temporarily create a new object on the stack and copy that one into the parameter. Finally we create a function return entry, invoke the function via `func_fun` and remove the return entry afterwards (see translation unit).

E_constructor Similar to `E_funcall`, but implicitly returns an object of its own type. First of all, we evaluate the expression of the return object (which is equivalent to the `this` pointer) and put it on top of the "this stack" (see `E_variable`). Then we get the parameters and allocate them on the stack. Now we take the arguments, evaluate them and initialize the allocated parameters with them or with their default values. A pointer to the object under construction, named `__receiver`, is put on the stack and initialized to point to the new object. Now the initialization list of this constructor is processed, either as a list of statements that are executed or as a list of argument expressions that are evaluated. Finally, we process the constructor body by calling the statement function with the `S_compound`

as its argument. Once all has been done we remove the top element of our "this"-stack, since we are leaving this constructor immediately.

E_fieldAcc This node has two child nodes: The field to access (which is a variable node) and the object (which is an expression node and might be again an `E_fieldAcc`). First, the visitor continues processing the object. Then, we test if the member is static. If this is not the case, we put the object and the member together to get the final dependency (See See Section 3.3.3 and object `b1` in Figure 3.17). Otherwise we just drop the enclosing expression and purely add a link to the static member alone.

E_unary, E_effect, E_binary These are straightforward to implement, we evaluate the expression (or in case of binary operations the left and the right one) and add a link(s) to the expression(s). A special case, however, is the comma operator that glues expressions together. We have to add only one link to the right expression, evaluating but ignoring the left one.

E_addrOf The only child node is an expression, so the visitor continues with that node. When coming back, the expression is used to find out which memory locations it refers to. Then an artificial pointer is put on the stack pointing to these locations and a link from this node to that pointer is added.

E_deref This node is the counterpart to `E_addrOf`. First, the expression is processed. Then, we determine which locations in memory this expression refers to. All these destinations must be pointers, failing otherwise. Since a pointer might point to more than one location, all are collected and this `E_deref` node will link to all of them. Special attention is required when dealing with indirect dependencies of pointers. Recall they might have been set conditionally, therefore a pointer not just points to some locations, but additionally might depend on (other) locations that influenced where it points (see Figure 4.2). Before processing a statement we reset a buffer collecting all these indirect dependencies. This buffer is used here to track dereference operations and to remember that information flows when using such a pointer (see Section 4.4, statement function).

E_cond The ternary operator can easily be replaced by some `if~then~else`, thus processing it is completely equivalent to that one of `S_if`.

E_assign The assignment operation is (besides the initialization of a variable) the only situation when the state of the program changes, i.e., when data is copied from one location to another. Therefore we process the source expression first, then the destination expression and finally we let the memory model perform the copy operation. This is explained in detail in Section 4.2

E_new After the type of the variable has been parsed, we allocate memory on the heap and add an artificial pointer that points to the location. The variable itself is placed on the heap, while the pointer is put on the stack.

E_keywordCast Casts are not supported, neither old C-style casts (`(int)3.0`) nor C++-style (`static_cast<int>(3.0)`). Only reinterpret casts from

one pointer type to another are allowed. All other casts are rejected (See Section 3.5, casts).

Not implemented are the remaining expressions: E_stringLit, E_cast, E_sizeof, E_sizeofType, E_delete, E_throw, E_typeidExpr, E_typeidType E_stringLit, E_cast, E_sizeof, E_sizeofType, E_delete, E_throw, E_typeidExpr, E_typeidType, E_grouping, E_arrow, E_statement, E_compoundLit, E__builtin_constant_p, E__builtin_va_arg, E_alignofType, E_alignofExpr, E_gnuCond, E_addrOfLabel.

4.5 Summary

At the beginning of this chapter we explained our choice of Ocaml. Then two interesting aspects of the implementation were explained in detail, namely how `if~then~else` is implemented and in which way we access the abstract memory. The third and longest part of this chapter dealt with the operations we perform on each individual AST node. The next chapter will show step by step how our tool processes small snippet of C++ code before we put the pieces together and check the ipc-short-cut of the Fiasco microkernel.

Chapter 5

Evaluation

In the following chapter we present the results of applying our tool to C++ code to automatically infer information flow. We analyzed more than 30 small artificial C++ programs to validate several relevant corner cases in C++. Then we present a case study analyzing the `ipc_short_cut()` of the L4-Fiasco microkernel.

5.1 Assign, Initialization and Temporary Flow

Very basic statements like assignments and variable declarations with an initialization expression are easy to handle and to verify. It is natural that when copying data from one variable to another information flows explicitly (see Figure 5.1). The first two statements (`int out1; out1 = in1`) are a declaration and an assignment. The third and fourth statement (`int out2 = in2; int out3(in3)`) are declarations with different kind of initialization expressions. The table on the right side shows the labels of the *@in* and *out* variables.

The second block of statements shows a piece of code that is not typeable with flow-insensitive type systems. `In_out` is both input and output variable, in the sense that it holds input data, potentially modifies it and returns it after program completion. Its initial value is copied to a temporary variable `tmp`, then `in_out` is written with the sum of `out1`, `out1` and `out3` and finally `tmp` is copied back to `in_out`.

From the memory state after program completion (see Figure 5.1, right table) we can see that `out1` depends on `in1`, `out2` on `in2` and `out3` on `in3` – as expected. The single row below the table shows `in_out`'s label between its first and second assignment: {1,2,3}. It depends on all three locations, but the last statement overwrites the label and replaces it with {4}, the final label of `in_out`. The temporary leakage of information got corrected before program termination, thus at the end no information is leaked through `in_out`.

5.2 If∼Then∼Else and the ?: Operator

The ternary operator `?:` is very similar to `if∼then∼else` statement. The only difference is, that `?:` is an expression and `if∼then∼else` is a statement. It is straightforward to replace `?:` with an equivalent conditional statement using a

```
int in1; // @in
int in2; // @in
int in3; // @in
int in_out; // @in @out
void main() { // @entry
    int out1; out1 = in1; // @out
    int out2 = in2; // @out
    int out3(in3);  // @out

    int tmp = in_out;
    in_out = out1 + out2 + out3;
    in_out = tmp;
}
```

location	type	name	label
1	int	in1	{}
2	int	in2	{}
3	int	in3	{}
4	int	in_out	{4}
6	int	out1	{1}
7	int	out2	{2}
8	int	out3	{3}

4	int	in_out	{1,2,3}

Figure 5.1: Test case 1: Assignment and initialization.

temporary variable as an intermediate – in fact, this is precisely what gcc does. Unfortunately, Elsa does not replace `?:` with a temporary variable, but handles it differently, which causes trouble in some specific situations. In our case this happened in `Thread * Thread::id_to_tcb(Global_id id)`:

```
return reinterpret_cast <Thread *>
  (id.is_invalid() || id.gthread() >= Mem_layout::max_threads() ?
  0:Mem_layout::Tcbs+(id.gthread()*Config::thread_block_size));
```

As a result an abstract storage location is going to be written that does not exist. If we replace the code above with the following, it compiles properly and is correctly type-checked.

```
unsigned int temp;
if(id.is_invalid() || id.gthread() >= Mem_layout::max_threads()){
  temp=0;
} else {
  temp=Mem_layout::Tcbs+(id.gthread()*Config::thread_block_size);
}
```

The only restriction is that the following cast (`reinterpret_cast <Thread *>`) is not allowed, we cannot cast an integer to a pointer, since the pointer would have to point to some abstract storage location, which is impossible if we use a given integer. The final configuration of the memory is not listed here, because it would be too long, instead 2 smaller examples are given.

Most of the time, constructs with `?:` are processed correctly, and if not, the tool will fail and we will have to rewrite the specific statement with `if`∼`then`∼`else`. The following code (Figure 5.2) is the first example demonstrating all 4 cases: direct (`input2` and `input3`) and indirect flow through if-guard (`input1`), and direct (`true`) and indirect (`in4`) flow through `?:`.

```
int in1; // @in
int in2; // @in
int in3; // @in
int in4; // @in
int out; // @out

void main() { // @entry
    int tmp;
    if( in1 ) {
        tmp = true ? in2 : in3;
    } else {
        tmp = in4 ? 0 : 1;
    }
    output = tmp;
}
```

location	type	name	label
1	int	in1	{}
2	int	in2	{}
3	int	in3	{}
4	int	in4	{}
5	int	out	{1,2,3,4,$\perp_b$,$\perp_i$}

Figure 5.2: Test case 2: Direct and indirect flow through conditional.

```
int in;  // @in
int out; // @out
int i,j;
void main() { // @entry
    int *p;
    if( in ) {
        p = &i;
    } else {
        p = &j;
    }
    if( p == &i ) {
        out = 0;
    } else {
        out = 1;
    }
}
```

location	type	name	label
1	int	in	{}
2	int	out	{1,$\perp_i$}
3	int	i	{}
4	int	j	{}
5	void	fun_main	{}
6	ptr {3,4}	p	{1}

Figure 5.3: Test case 3: Set a pointer conditionally and dereference it.

An interesting scenario is conditionally setting pointers and then compare them. First we set a pointer and then compare it with well-known addresses. Figure 5.3 shows such a case. In the first if-block, the pointer `p` is set depending on the value of `in` to point either to variable `i` or `j`. This is used in the second if-block by comparing `p` with the address of `i`. Thus we have an indirect information flow through the information where `p` points (either to `i` or to `j`). This is correctly modelled, since `p`'s type is ptr{1}, indicating that this pointer depends on location 1, which is `in`. So `out` correctly depends on `in` and on some integer literals (`0` and `1`).

5.3 Functions and References

A very basic function is the identity function: Taking an argument and simply returning it. For example `output = id_function( input )` does exactly this, copying input to output. Information flows from the argument (`input`) through the function to the expression that caused the function call (an `E_funCall` expression). Finally, the assignment copies the data to the destination, `output`.

If we replace on parameter by a reference, shown in Figure 5.4, then data is directly copied, without an `E_funCall` expression in between. Since `b` is a reference parameter pointing to `output`, when in the function body `a` is copied to `b`, `output` is directly written.

```
int input;  // @in
int output; // @out

void foo( int a, int& b ) {
    b = a;
}

int main() { // @entry
    foo( input, output );
    return 0;
}
```

location	type	name	label
1	int	input	{}
2	int	output	{}
3	int	a	{1}
4	ref{2}	b	{}

1	int	input	{}
2	int	output	{1}
3	int	a	{1}
4	ref{2}	b	{}

Figure 5.4: Test case 4: Function with a reference parameter. On the right side is the memory state immediately before and after calling the function.

The example in Figure 5.5 shows how a function returns a reference. `Foo` has two reference parameters and takes a third argument to decide which of the two should be returned. The statement that assigns `in2` to the returned reference has the function call as its l-value. Since one of the two parameters might be returned, this assignment is a write operation on two location, doing weak updates on both of them.

```
int in1, in2; // @in
int out1, out2; // @out

int& foo( int& x,
          int& y, flag ) {
    if( flag ) {
        return x;
    } else {
        return y;
    }
}

foo( out1, out2, in1 ) = in2;
```

location	type	name	label
1	int	in1	{}
2	int	in2	{}
3	int	out1	{1,2}
4	int	out2	{1,2}
6	int	flag	{1}
7	ref{3}	x	{}
8	ref{4}	y	{}
9	ref{3,4}	E_funCall	{1}

Figure 5.5: Test case 5: Function returning a reference that will be written.

Other interesting scenarios not discussed here are branching on conditionally set pointers or references (p/r), weakly updating p/r, again with conditionally created p/r's, or dereferencing p/r's a couple of times, both within r-values and l-values, gathering their dependencies or the locations where they point.

5.4 Objects and Member Functions

The last missing pieces are classes and objects. As described in Section 3.3.3, classes consist of their member variables which we model with complex locations. For instance, an object at location x with a member having an AST-ID y will get the abstract storage location $x.y$. Figure 5.6 covers the following aspects:

- references to fundamental data and to objects,
- default parameters (both fundamental and objects),
- member initialization through expression and statement list,
- implicit `this` usage,
- member function invocation, and
- the =operator to copy objects.

In the given example we have two classes: An `int_wrapper` simply containing an integer and an `int_pair` encapsulating two `int_wrapper`s, each with its own data member. The first statement of the main function declares an `int_wrapper`, initializing its member with `in`. The next statement declares another `int_wrapper`, this time using the default parameter to initialize its member. Finally we construct an `int_pair` out of the two `int_wrapper`s. Since the members of an `int_pair` are objects, they cannot be initialized by expressions, thus a list of statements is used instead. The next statement declares a second `int_pair` named `q`, without giving any arguments, so the default arguments of `q`'s constructor are used (two `int_wrapper`'s, initialized with some default values, here 23 and 42). In order to see if the `=operator` is correct, we assign `p` to `q` in the following statement. And in the last statement besides the `return 0`, the sum of `i` and `j` is calculated by calling `q.sum()`, which in turn calls `fst.get()` and `snd.get()` of its members and adds them.

The table on the next page shows, that information has flown from location 1 (variable `in`) to location 2 (variable `out`). The second dependency($\perp_i$) is the integer literal `0` that belongs to `j`. The complete final memory state of this example contained 48 entries. We omitted temporary objects, artificially injected pointers/references and variable `q` to improve readability.

location	type	name	label
1	int	in	{}
2	int	out	$\{1,\perp_i\}$
4	cmp int_wrapper	i	{}
4.52	int	i.data	{1}
7	cmp int_wrapper	j	{}
7.52	int	j.data	$\{\perp_i\}$
8	cmp int_pair	p	{}
8.273	cmp int_wrapper	p.fst	{}
8.273.52	int	p.fst.data	{1}
8.274	cmp int_wrapper	p.snd	{}
8.274.52	int	p.snd.data	$\{\perp_i\}$

5.5 ipc_short_cut()

In the foregoing sections we evaluated small pieces of artificial code, like assignments, `if`∼`then`∼`else`-branching, references or member functions. Now we put together all the pieces and evaluate a real-world scenario, the ipc-short-cut from the Fiasco microkernel. Some adaptations were required to get it checked. On the one hand we had to set up an appropriate environment for the function call: two threads that communicate, a sender and a receiver list, two UTCB's, the receiver state and so on. On the other hand some modifications became necessary to overcome limitations of our tool. Assembly code needed replacement, templates had to be instantiated and last but not least Elsa showed a strange behavior, so that we had to rewrite some minor parts to get it compiled correctly. Elsa implements the `?:` operator differently than gcc, causing some problems regarding temporary variables. A subtle bug happened when the return type of a function differs from the type of the expression returned. If a constructor exists to do the cast, it would be invoked implicitly, but Elsa calls the copy constructor, which leads to type errors later on. Therefore we added a temporary object to do the cast explicitly. The following list of modifications is not complete, but lists the most important changes.

- We had to remove `context_of()` and `id_to_tcb()`, because there was no way to find a sound typing, these two functions do really bad magic (e.g. calculating a TCB from a stack pointer).
- We removed some casts and added more temporary objects when implicate object castings occurred (e.g. `return L4_snd_desc( _eax )` instead of `return _eax`).
- Two thread objects (sender and receiver) have been added to get rid of `id_to_tcp()`
- To simulate the sender and receiver stack frame we added an `Entry_frame` and a `Sys_ipc_frame` and filled it with the corresponding registers.
- Assembly code doesn't work yet and needs to be removed or replaced, like in `memcpy_mwords()` to copy the `UTCB`, in `cli()` and in `interrupts()` or `current()`. The memcopy has been replaced by 32 assignments, the interrupt flag became a variable `current()` always returned the sender thread, assuming that no scheduling occurs within the ipc-short-cut.

```
int in;  // @in
int out; // @out

class int_wrapper {
public:
    int_wrapper( const int& i=0 ) : data( i ) {}
    int get() { return data; }
private:
    int data;
};

class int_pair {
public:
    int_pair( const int_wrapper& i=int_wrapper(23),
              const int_wrapper& j=int_wrapper(42)
            ) : fst( i ), snd( j ) {}

    int sum() { return fst.get() + snd.get(); }
private:
    int_wrapper fst;
    int_wrapper snd;
};

int main() {   // @entry
    int_wrapper i( in );    // member init in initialization list
    int_wrapper j;          // init list with default argument
    int_pair p( i, j );     // init members through statements
    int_pair q;             // default args
    q = p;                  // =operator
    out = q.sum();
    return 0;
}
```

Figure 5.6: Test case 6: Complex scenario with objects, member functions, constructor calls, default parameters, initialization lists and an =operator.

- Templates needed to get instantiated by hand, since they are not supported yet (e.g. `nonull_static_cast` or `sys_frame_cast`).
- `Thread::copy_utcb_to_utcb()`: we removed `transfer_fpu()` and related code as well as `copy_ts_to_utcb()` and `copy_utcb_to_ts()`, because that got too complex.
- Finally, the scheduling at the end of the short cut (`switch_exec_locked()` and `ready_enqueue()`) was ignored, we added empty bodies to the functions, since otherwise we would have to check the whole scheduling code as well, something we didn't want to do.

With all these modifications we have the following situation: two arbitrary, but fixed threads A and B are involved in the IPC system call. Thread A invokes a send operation to B, while B is doing an open or closed receive operation with A in the set of its communication partners.

After we analyzed the data flow in `ipc_short_cut()` the memory model contained 567 entries, more than the half of them were results from function calls, artificially injected pointers (e.g., `this` or `E_addrOf`) and references related to constructors. The second half covered temporary objects, local variables and finally input/output variables, like sender and receiver thread, their UTCB's, their register files and their stack frames. The interesting parts (sender, receiver, UTCB's) are shown in Figure 5.7 (recall: negative labels are constants).

As you can see, the sender stack frame depends linearly on the input register file, except the eax register. The entries for cs, csu, ss and ssu are not shown, but also depend solely on the corresponding register. The location [31 4849], which is the eax field of the `Entry_frame`, received its data from [20], the input register ecx, which is the timeout parameter and from [25], the ebp register holding the receive descriptor. This is precisely what we would expect and what the L4 Reference Manual[Lie96] states: Both, the timeout and the receive descriptor, flow into the return value of the `ipc_short_cut()` system call. Other data flows, besides constants, do not take place.

The second part of the table shows the receiver side and how data flows there. Register ecx stays untouched, it's just the receive timeout. The sender ID is taken from the esi register of the sender (location 22) and appears in most of the labels of the receiver `Sys_ipc_frame`. To simulate a correct environment for the ipc-short-cut, we needed an additional `sender_id` of type `L4_uid`, which should correspond to the sender's esi register. This is the reason why `f._esi` depends on [62.10] instead of [22]. The payload of this system call, two 32bit words transmitted from the sender to the receiver, are allocated in register ebx and edx. And indeed, receiver registers ebx and edx got their data from the sender's ebx and edx registers.

The third interesting observation are related to the two UTCB's, which carry additional payload that does not fit into the two registers. The copying was originally done by a fast `rep movsl` assembler instruction. We replaced it with 32 single assignments. The table shown does not contain all 32 entries, only the first and the last, but as we can clearly see do all 32 receiver words correctly depend on the 32 sender words ([63.3024] $\rightarrow$ [65.3024] $\cdots$ [63.3272] $\rightarrow$ [65.3272]), plus location 22, the sender ID. The -3 appearing in most labels means that some constants also contributed to the values, but constants do not reveal any information.

location	type	name	label
sender register file and stack frame			
17	int	eip	{}
18	int	eflags	{}
19	int	esp	{}
20	int	ecx	{}
21	int	edx	{}
22	int	esi	{}
23	int	edi	{}
24	int	ebx	{}
25	int	ebp	{}
26	int	eax	{}
31	cmp Entry_frame	e	{}
31.4854	cmp Syscall_frame	e	{}
31.4800	int	e._ecx	{20}
31.4809	int	e._edx	{21}
31.4817	int	e._esi	{22}
31.4825	int	e._edi	{23}
31.4833	int	e._ebx	{24}
31.4841	int	e._ebp	{25}
31.4849	int	e._eax	{-3, 20, 25}
31.5299	cmp Return_frame	e	{}
31.5254	int	e._eip	{17}
31.5274	int	e._eflags	{18}
31.5282	int	e._esp	{19}
receiver register file and stack frame			
35	int	ecx2	{}
36	int	edx2	{}
37	int	esi2	{}
38	int	edi2	{}
39	int	ebx2	{}
40	int	ebp2	{}
41	int	eax2	{}
42	cmp Sys_ipc_frame	f	{}
42.4854	cmp Syscall_frame	f	{}
42.4800	int	f._ecx	{35}
42.4809	int	f._edx	{-3, 21, 22}
42.4817	int	f._esi	{62.10}
42.4825	int	f._edi	{-3, 22, 23}
42.4833	int	f._ebx	{-3, 22, 24}
42.4841	int	f._ebp	{40}
42.4849	int	f._eax	{-3, 22, 26}
62.10	int	sender_id._raw	{}
sender and receiver UTCB			
63	cmp Utcb	sender_utcb	{}
65	cmp Utcb	receiver_utcb	{}
65.3024	int	receiver_utcb.value_0	{-3, 22, 63.3024}
...	...	...	...
65.3272	int	receiver_utcb.value_31	{-3, 22, 63.3272}

Figure 5.7: Final memory configuration after returning from ipc_short_cut()

5.6 Summary

This chapter started with the evaluation of small pieces of C++ code. We have seen how information flow evolves. Beginning with assignments, we saw how conditionals are handled, had a look at pointers and references and their peculiaritiesa, analyzed functions and finally came to objects with their member functions.

Once we finished all the experiments, the touchstone for our approach was a piece of kernel code taken from the Fiasco microkernel, namely the `ipc_short_cut()` function. The whole analysis consisted of about 20 classes and covered roughly 700 lines of code. We encountered about 200 variable declarations, 100 function calls and 78 constructor calls. We have successfully shown which information flow is caused by invoking this system call. What we found is conform to the description stated in the L4 Reference Manual.

Chapter 6

Conclusions and Future Work

We presented an approach to statically check data flow in C++ programs. Our ultimate goal is to provide tool support for proving non-interference. Our tool allows this by precisely calculating how data flows through a given program. We applied the idea of dynamically labeling all program variables with additional information describing from where their values originated. The type system we used is flow-sensitive and capable of handling objects, function calls as well as reference and Java-style pointers.

To show the practical usage of the presented approach we implemented a tool that iterates through an abstract syntax tree, extracts data dependencies and updates a memory model so that information flows become visible. Since we support a rich subset of C++, the whole analysis was done automatically. We first tested the tool with several artificial pieces of C++ code in order to assert its correctness and then applied it to an optimized path of the IPC system call of the Fiasco microkernel, namely the `ipc_short_cut()` function.

Some minor modifications to the `ipc_short_cut()` function were required (see Section 5.5), like replacing assembly code by equivalent C++ code. We also had to provide an environment in which the system call can be simulated, since everything happened statically at compile time, thus no "real" objects existed.

We have shown that the detected information flow is consistent with the specification in the L4 Reference Manual [Lie96]. Data is transmitted from the sender to the receiver thread, either in the two specified registers or via their UTCBs (user thread control blocks) and data does not leak to any other than the intended locations.

We have successfully used the ideas of dynamic labeling [JPW05] and of universal lattices [HS06] and combined them with the practical approach of CQual[FTA02] and Oink [DSWM]. By analyzing the `ipc_short_cut()` function of the IPC system call we demonstrated the usability of our tool and showed that the occurring information flow is correct.

6.1 Future Work

While developing and improving our tool to cover more C++ constructs we had to make some decisions about the design that led us to new question. One of the first things is to refine the memory model by adding information about the size and the layout of memory cells as well as precise type information, maybe even the type description from the abstract syntax tree. If we see a abstract memory location belonging to an object, we will then know the type of that object and therefore its memory layout. Knowing this allows us to support more complex memory accesses, like reading an int where 4 characters are, bit manipulations and basic pointer arithmetic and casts.

Information about how memory cells are relatively located to each other, e.g. in arrays, will help us to support pointer arithmetic to some extend. Modelling pointers more precisely and relying on the programmer to use only reasonable pointer arithmetic will increase the range of checkable programs. Other directions we would like to explore are loops, casts and templates, These directions are outlined in Section 3.5.

Finally, a challenging research is the combination of flow-insensitive and flow-sensitive type systems. Having the advantages of flow insensitivity (like typing loops easily) and the strengths of flow-sensitive typing, especially its accuracy regarding alias analysis. A combination of both might yield promising results.

Bibliography

[C] ISO/IEC FDIS 14882:1998(E).

[CW07] Karl Chen and David Wagner. Large-Scale Analysis of Format String Vulnerabilities in Debian Linux. In *PLAS '07: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 75–84, New York, NY, USA, 2007. ACM.

[DSWM] Karl Chen Daniel S. Wilkerson and Scott McPeak. Oink. http://www.cubewano.org/oink.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 192–203, New York, NY, USA, 1999. ACM.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2002. ACM.

[Hor97] Susan Horwitz. Precise Flow-Insensitive May-Alias Analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, 1997.

[HS06] Sebastian Hunt and David Sands. On flow-sensitive security types. *SIGPLAN Notices*, 41(1):79–90, 2006.

[JPW05] B. Jacobs, W. Pieters, and M. Warnier. Statically checking confidentiality via dynamic labels. In *WITS '05: Proceedings of the 2005 Workshop on Issues in the Theory of Security*, pages 50–56, New York, NY, USA, 2005. ACM Press.

[Lie95] J. Liedtke. On μ-Kernel Construction. *SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[Lie96] Jochen Liedtke. L4 Reference Manual - 486, Pentium, Pentium Pro, 1996.

[Mye99] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, New York, NY, USA, 1999. ACM.

[NCH+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[NMW] George Necula, Scott McPeak, and Wes Weimer. Taming C Pointers.

[SM03] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 2003.

[Smi01] Geoffrey Smith. A New Type System for Secure Information Flow. In *CSFW '01: Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, pages 115–125, Washington, DC, USA, 2001. IEEE Computer Society.

[VS97] Dennis M. Volpano and Geoffrey Smith. A Type-Based Approach to Program Security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.

[Wei03] Westley Weimer. The CCured Type System and Type Inference. Technical Report UCB/CSD-03-1247, EECS Department, University of California, Berkeley, Dec 2003.

[Wu08] Qiang Wu. Survey of Alias Analysis; http://www.cs.princeton.edu/~jqwu/memory/survey.html, 2008.