INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

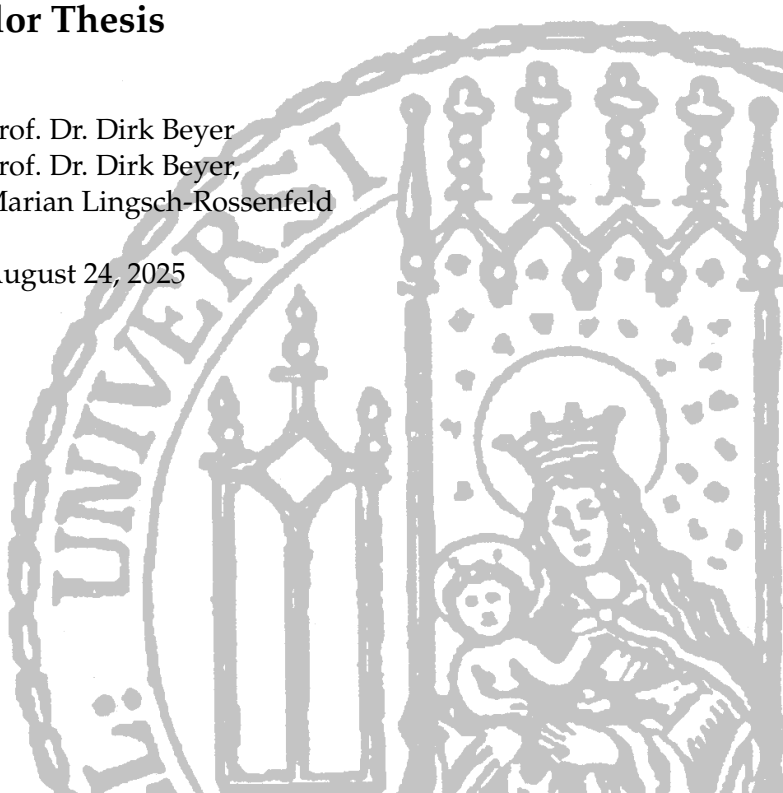
TAINT ANALYSIS IN CPACHECKER

Diego Ignacio Salgado Esparza

Bachelor Thesis

Supervisor Prof. Dr. Dirk Beyer
Advisor Prof. Dr. Dirk Beyer,
Marian Lingsch-Rossenfeld

Submission Date August 24, 2025



Statement of Originality

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments. For some grammatical corrections, I have used JetBrains AI Assistant, powered by the AI model GPT-4o.

München, August 24, 2025 Diego Ignacio Salgado Esparza

Abstract

Ensuring software security is essential to protect sensitive information from unintended disclosure or adversary exploitation. Taint analysis, a method grounded in information flow analysis is an effective technique to track sensitive data through programs, enabling the detection of vulnerabilities such as unauthorized information disclosure. This thesis presents the implementation and evaluation of a symbolic taint analysis module for CPAchecker, a configurable software verification framework. The implemented module systematically tracks the flow of sensitive information through C programs, identifying potential leaks and vulnerabilities in a set of synthetic benchmark programs. For the benchmarking process a combination of a set of test cases created alongside the development of the analysis implementation, and other set of benchmarks from external sources. In addition, all benchmarks were formatted for consistency. The outcomes demonstrate the robustness and extensibility of the developed module, but also show that there is space for future work on the aspects that could not be covered within the scope of this work.

Acknowledgments

I want to express my sincere gratitude to my supervisor Marian Linsch-Rossenfeld for his guidance and feedback during the course my process of writing this thesis, for sharing with me his knowledge and valuable insights about the CPAchecker, Computer aided Verification and the main topic of this thesis; Information-flow and Taint Analysis.
Additionally, I would like to mention that the first steps of the implementation of the taint analysis were made by Jan Haltermann. He made a well-written prototype that contained the necessary classes and basic methods that the analysis needs to be used as a CPA in CPAchecker but was not yet a working implementation. As part of this thesis, I expanded that first prototype so that it can analyze C programs, tracking information flow and reporting taint violations.

Contents

List of Figures

CHAPTER 1

Introduction

Ensuring software security has become an increasingly critical challenge, as modern systems frequently handle sensitive information such as personal data, authentication credentials, medical records, financial information, or classified materials. Examples of critical software security breaches include vulnerabilities like SQL-Injections, leaks of personally identifiable information through improperly sanitized inputs, and unauthorized disclosure of cryptographic secrets [3]. A technique to address this is **taint analysis**, a formal method based on **information-flow analysis** used to track the flow of sensitive data through software programs. By marking sensitive data (*tainting*) at specific sources and observing its propagation throughout a program, taint analysis can help detect when such data reaches insecure destinations, also known as sinks. Taint analysis enables developers to identify potential vulnerabilities, including unintended data leaks and unauthorized access paths.

The goal of the taint analysis is to determine whether at the end of program execution there are secret variables that are **adversary-observable**. The **adversary** is the entity that attempts to obtain information about confidential data from the execution of a program. In this work, we use a threat model in which the adversary can observe the *public* (*low-security*) outputs after the execution of a program that operates on confidential data [7]. A *private/secret* or *high-security* variable is, in contrast to a public variable, a variable that carries sensitive information that must not be adversary-observable. The information flow analysis tracks the influence of the different program variables on each other during program execution. In the context of detection of unintended information disclosure, the influence of high-security variables on low-security variables is particularly interesting and critical, because even if after the program completion a secret variable is not directly leaked, an adversary could gain partial or full knowledge about the confidential information through a public output that was influenced by a high security value during the program execution. Taint analysis flags variables that are influenced by high-security values

as **tainted** and transitively propagates the taint with each interaction (assignments and assumptions) between the variables.
This thesis presents an implementation of **symbolic taint analysis** and an evaluation of its performance over a set of benchmarks in form of C-programs with predefined assertions about the taint status of the program variables at certain points of the program execution (ground truth).

Our taint analysis is implemented as a *Configurable Program Analysis* (CPA) [1] of the **CPAchecker** [1] framework, a state-of-the-art tool for software verification which will be briefly explained in Chapter 3.1. The benchmarking of the analysis is made with the benchmarking tool **BenchExec** [2]. Both tools are open-source and developed at the chair of Software and Systems at the Ludwig Maximilians Universität in Munich. (poner links a los repositorios al pie de la página)

Taint analysis in CPAchecker was implemented as a continuation of an initial prototype written by Jan Haltermann. The prototype was not yet a working implementation, meaning that it was not yet able to analyze a C program. However, it was a solid foundation that counted with the necessary classes (CPA, Transfer-Relation and State objects) to handle state generation, which were used in this thesis as base for the implementation of the actual analysis and its posterior benchmarking.

The thesis consists of 6 chapters. To this chapter follows Chapter 2 that introduces the concepts of *information-flow* and *taint analysis*, their relation and their application in the implemented analysis in CPAchecker. Chapter 3 explains the aspects of CPAchecker that are relevant for the analysis and how they are going to be used for implementing and benchmarking our taint analysis. Chapter 4 goes deeper into the actual implementation details of the taint analysis in CPAchecker, also explaining the challenges and its logic, e.g., behind the state generation strategies and other important aspects of the analysis. Chapter 5 shows the benchmarking process and the result of our analysis. Chapter 6 presents the conclusion of this work and suggests some ideas to expand it in the future.

The main contributions of this thesis are:

- **Implementation of Taint Analysis**: Development of a simple (static + single copy + basic threat model), yet complete and expandable prototype for a taint analysis module for CPAchecker, offering precise and systematic tracking of information flows in C programs.
- **Benchmark Design**: Creation and collection of a suite of controlled test cases to reliably evaluate the precision of the implemented taint analysis. All benchmark programs from external sources were be put in a common format.
- **Performance Insights**: Experimental validation of the taint analysis module's effectiveness in the benchmark set.

[1] https://gitlab.com/sosy-lab/software/cpachecker
[2] https://github.com/sosy-lab/benchexec/blob/main/doc/INDEX.md

CHAPTER 2

Related Work

Taint analysis and information flow have been studied extensively, as these approaches play a crucial role in software security by identifying unintended data leaks and vulnerabilities. Yang et al. [7], in their work on lazy self-composition for security verification, introduced a formal methodology to analyze information flow within programs. Their approach employs a more complex verification mechanism compared to the one implemented in this thesis, as it relies on lazy self-composition to define and track information flow. However, it provides valuable theoretical insights into secure program analysis, such as formal definition of the information flow problem and symbolic taint analysis, which were used in this work.

CPAchecker, used as the foundation for this implementation, is a state-of-the-art software verification framework that integrates SMT-based reasoning and configurable program analysis, as presented in the work "A Unified View on SMT-Based Software Verification." [1]. This framework handles critical aspects of verification, such as parsing, state exploration, and abstraction management, which in general allowed the focus of this thesis to remain on the design and evaluation of the taint analysis module.

The work "Timing side-channel attacks" [6] of Martin Schwarzl et al. highlights how even subtle, indirect interactions between data can lead to critical vulnerabilities, which underscores the importance of developing accurate tracking mechanisms for both explicit and implicit data flows. Understanding the importance of indirect flows was essential to shape the taint propagation strategy of this thesis.

Historical perspectives also offer valuable insights into information flow verification. A foundational work, "Certification of Programs for Secure Information Flow" by Dorothy and Peter Denning [2], introduces one of the earliest compile-time mechanisms for securely certifying information flow in programs. Despite being an older publication, it remains relevant today, particularly in its structured

handling of information flows across data structures. The clear definitions and theoretical underpinnings inspired some aspects of the analysis performed in this thesis.

CHAPTER 3

Information-flow and Taint Analysis

3.1 Information-Flow Analysis

Information Flow is a property that checks the explicit or implicit influence of *high security (private)* values on *low security (public)* values on a program [7].
Here, *explicit information flow* represents direct assignments or operations where tainted data is directly used, and *implicit information flow* refers to the propagation of influence through control-flow structures like conditional statements [2].

While the importance of explicit information flow is evident, implicit leaks can appear less critical. However, the importance of implicit information flow lies in the fact that confidentiality must ensure that sensitive data cannot be inferred by observing side effects, such as flows, execution time, or exceptions.

Among other applications of this technique, inspection of such relationships can deliver critical information about vulnerabilities in a system, addressing the *secure information flow problem*. The secure information flow problem encompasses challenges surrounding *data confidentiality*, which ensures that no unauthorized party gains visibility over private or high-security information, and *data integrity*, which ensures that high-integrity processes are not manipulated by lower-integrity or untrusted data [5].
Figures 1 and 2 illustrate the implicit and explicit information flow. Listing 1 shows an example of explicit information flow in a C program. We consider the variable `high_security_variable` to be a source of sensitive information, which due to the assignment `a += high_security_value;` in code line 3 becomes adversary observable through the variable `a` at the end of the program execution.
On the other hand, in listing 2 the variable `a` does not directly contain sensitive information, but its value is controlled by the `high_security_value`. An attacker can then gain information about high-security data even if it is not an explicit leak. This

is an example of implicit information flow. Other forms of implicit information flow are *Timing Side Channels* [6] and the mere presence of an error message, among others.

```
1 int main(int high_security_value) {
2     int a = 4;
3     a += high_security_value;
4     return a;
5 }
```

Listing 1: Explicit information flow from a high security variable to a low security variable

```
1 int main(int high_security_value) {
2     int a;
3     if (high_security_value) {
4         a = 1;
5     } else {
6         a = 0;
7     }
8     return a;
9 }
```

Listing 2: Implicit information flow in which a high security variable controls the state of a low security variable

3.2 Symbolic Taint Analysis

Taint analysis is a form of *information flow analysis*. In contrast with other forms of information flow analysis that could, for example, use execution of copies of a program with modified sensitive data in control structures to check whether the observable values were influenced, as shown in "Finding Information Leaks with Information Flow Fuzzing" [4], in taint analysis the core idea is to flag sensitive information as tainted, and track its propagation and interaction with other program variables, to determine whether at the end of the program execution, adversary observable variables are influenced by high security data. Here, it is also important to note that not only variable assignments (explicit taint) can be relevant for the taint propagation, operations inside control structures such as conditions in branches or loops are relevant as well (implicit taint).

We distinguish *symbolic* from *dynamic* taint analysis. In the first, the program is not analyzed at run-time and considers all possible inputs and paths that static verification allows. Dynamic analysis, on the other hand, checks only explicit paths of a program at run time. Since our implementation of taint analysis extends the capabilities of CPAchecker, a framework for static software verification, it is a symbolic analysis.

To describe taint analysis formally, we use the definition of symbolic taint analysis from the paper *Lazy Self Composition for Security Verification* [7].

We capture the taint of a formula φ over a signature Σ via

$$\Theta(\varphi) = \begin{cases} \text{false} & \text{if } \varphi \cap X = \emptyset \\ \bigvee\limits_{x \in \varphi} x_t & \text{otherwise} \end{cases}$$

Here, the signature represent the union of literals and program variables, and the formula φ can be a statement, such as `x + 1`, or an assumption (like `x < 1`), but note that φ is defined over the program signature Σ and therefore not necessarily over program variables; it could also be a statement over constant values, which are not a source of sensitive information, and therefore are never tainted. The symbol x_t is of sort boolean and says whether the program variable $x \in X$ is tainted or not. In our analysis, it can be expressed as $x_t \Leftrightarrow x \in T(\subseteq X)$, where T is the set of tainted variables. The state update $\widehat{T}r(X_t, X_t')$ of the taint status of the program variables X is then defined in Yang. et. al [7] as

$$\bigwedge_{x_t \in X_t} x_t' = \underbrace{\Theta(cond)}_{\text{implicit taint flow}} \vee \underbrace{(cond\ ?\ \Theta(\varphi_1) : \Theta(\varphi_2))}_{\text{explicit taint flow}}$$

This means that taint propagation can be triggered either by a direct assignment (explicit taint propagation) or when an assignment is controlled by a tainted variable (implicit taint propagation), e.g., when the condition of an if-statement contains a tainted variable and its body contains a variable assignment, even if the tainted variable is not involved in the assignment.

The intuitive behavior of the analysis that the formal definition above models can be resumed as: "For given program variables x_1, x_2 with x_1 private and x_2 public, taint may propagate from x_1 to x_2 either when x_2 is assigned an expression that involves x_1 or when an assignment to x_2 is controlled by x_1". This definition will be the basis for the taint propagation of our analysis and for defining the *information-flow violation* that the analysis will check.

The resulting behavior of the taint analysis is an overapproximation of the information flow, leading to some conceptual challenges such as false-positive alarms. Consider the program in figure 3. Intuitively one would think that the program variable `b` should become sanitized by the subtraction of the `high_sec` variable. For taint analysis this is not the case, since , as described above, for an assignment, the taint flows from the **right-hand side** (RHS) to the **left-hand side** (LHS). Taint analysis tracks the taint flow between program statements independently of their actual values, and therefore, the actual value of `b` is irrelevant for the taint analysis. This is an example of a **false positive** since `b` clearly does not carry the value of `high_sec` after the subtraction, but despite it, it will be flagged as tainted.

```
1 int main(int high_sec) {
2     int b = 1;
3
4     b =+ high_sec; // taint flows to b
5     b =- high_sec; // b is does not get sanitized
6
7     return b; // b is tainted
8 }
```

Listing 3: Overapproximation leads to false positive

CHAPTER 4

CPAchecker

CPAchecker [1] is a highly configurable SMT based software verifier developed at the chair of *Software and Systems* of the *Ludwig Maximilians Universität München*. CPAchecker supports verification of *C* and Java Programs, although the verification of Java programs is relatively new compared to the one for C, and therefore not yet as mature as the for C programs, which is currently completely robust and has been used for finding bugs in real-world programs e.g. in the Linux Kernel. Since the work in this thesis is based on analyzing C programs, our focus stays on the key aspects of CPAchecker's capabilities for the language C.

CPAchecker provides a highly adaptable modular framework based on *Configurable Program Analysis (CPA)*, a unifying foundation for various program analyses. Between others, it supports approaches such as *predicate abstraction*, *bounded model checking*, and *lazy abstraction*, which leverage SMT (Satisfiability Modulo Theories). The taint analysis module extends CPAchecker using symbolic abstraction techniques and configurable components to define sensitive sources, tainted and untaitned variables, and checks for assertions over the taint state of variables on certain points of the program execution, among others. The module leverages CPAchecker's abstraction capabilities to propagate taint information through C programs.
Since almost all the concepts around information flow can be better understood by looking at the CFA of a program, we take a look at its definition and role in CPAchecker.

4.1 Control Flow Automaton

A *Control Flow Automaton (CFA)* $A = (L, l_{init}, G)$ is a structure used to represent a program in form of a graph. L is the set of program *nodes* and G the set of program *edges*, with $l_{init} \in L$ being the initial node (entry point) of the program, and the edges $g \in G$ being 3-tuples with $g = (l_i, op, l_j)$ for the node $l_i, l_j \in L$,

and an operation *op* which can be either an *assignment* or an *assume* operation over the set of program variables X, applied when transiting from node l_i to node l_j in the edge g. CPAchecker generates a CFA for the analyzed programs, which can be very useful, among others, to visually understand the information flow in a program. Figure 4.1 shows a C code snippet and its CFA representation.

```
1  int main(int tainted) {
2      int a = 2;
3      int x;
4
5      if (tainted < 10) {
6          x = a;
7      } else {
8          x = tainted;
9      }
10
11     return x;
12 }
```

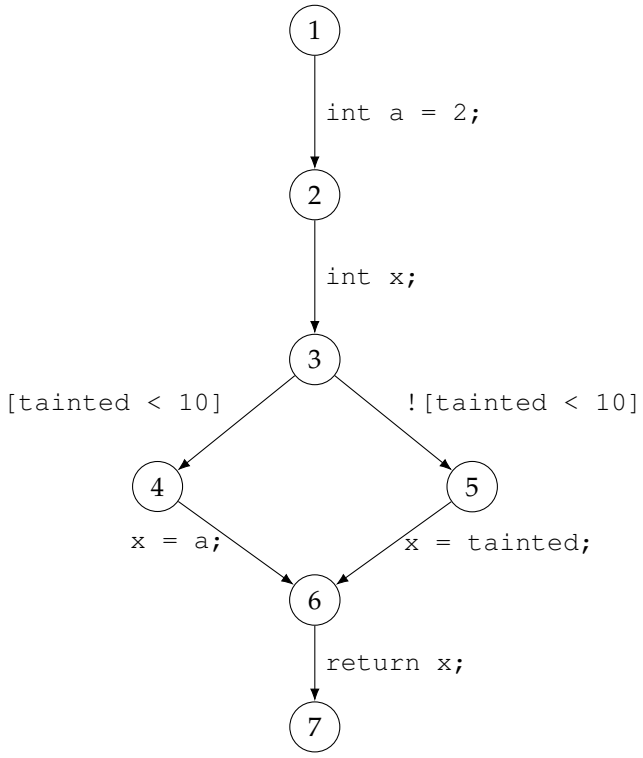


Figure 4.1: C code on the left and its corresponding CFA on the right

CPAchecker allows the tracking of the CFA on each step of the CPA algorithm. It parses the content of the given programs, e.g., program variables, assignments, declarations, etc., and all this meta data is then contained in the CFA and passed to the analysis. When writing or debugging an analysis in CPAchecker, a developer can count with a robust tracking of the CFA of the analyzed program at each step of the algorithm, being able to see all the important information, such as the type of edge, the entering and leaving nodes, and the statements that are present at specific points of the program execution. A major part of the implementation of an analysis consists of handling the different types of edge and the statements on each edge.

4.2 CFA Edges

There are several types of edges that CPAchecker recognizes. Every type of edge contains information about its predecessor and successor nodes, and each node contains

information about its leaving and entering edges, so that outgoing from one edge one can reconstruct the whole CFA. Additional information like the source lines, raw statement and file location that correspond to the edge can be found in every type as well. However, each particular type of edge has additional specific attributes.

In the following we focus on the types of edge that are relevant for our implemented analysis, to set the ground for the explanation of the taint analysis in Chapter 4. Consider $x \in X$ to be an arbitrary program variable in the set of program variables of a C program.

1. **Assume Edge**: An assume edge represents the check of a condition inside some flow control structure in the program, such as if-statments or loop-conditions. It consists of a plain condition, like `x > 3`. Because of that, the predecessor node of an assume edge always has two leaving edges, one for each branch, as figure 4.2 shows.

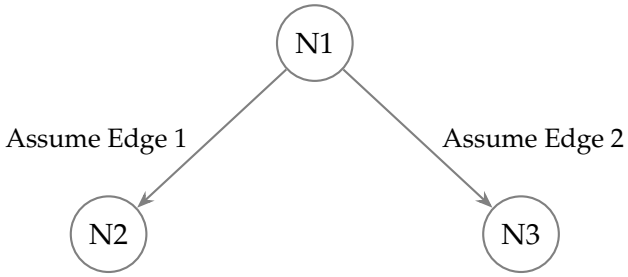


Figure 4.2: Node N1 is the predecessor node of assume edge 1. N1 has two leaving edges

   Since this kind of edge determines the branching of the CFA, it plays a critical role in a path-sensitive analysis, like our implementation of taint analysis. This edge contains additional information related to the given condition, such as the expression it self, whether it is swapped, and a truth assumption, which indicates which branch of the condition is being explored.
   Note that in *C*, unlike other programming languages like Java, the logical conditions are not necessarily treated as boolean variables. In general, in *C* a value is equivalent to `false` when it is equal to 0, and `true` otherwise. Therefore, for example, an assume edge created from the line `if(x)...` is totally valid, and internally means, for CPAchecker, to check whether `(x == 0)` or `(!(x == 0))` holds.
   The special case of assumptions in which `x` is initialized as a *non-deterministic* variable that does not hold an explicit value are valid as well, and its handling represent an interesting case for the taint analysis.

2. **Declaration Edge**: Variable declarations without an initializer like `char x;`, or with an initializer like `int x = 4;` or function declarations like `void`

`foo();` are examples of statements that correspond to a declaration edge. This kind of edge contains additional information about the declaration, such as the (nullable) *initializer*, the name, scope and data type of the declared variable.

3. **Statement Edge** : A statement edge can represent either a simple *expression*, an *expression assignment*, a *function call*, or a *function call assignment*. Calls to *external functions* and *C built-in functions* like `sizeof` or `alignof` are handled as statement edges, but fully implemented functions that are defined outside the scope of the main function, are handled as a *function call edge* instead.
   Simple expressions like `var` without any initialization or declaration are not particularly relevant for the analysis, but are necessary for the generation of new states to prevent an early termination of the analysis execution. Expression assignments and function call assignments consist of a *left-hand side* (LHS) and a *righ-hand side* (RHS), which are expressions themselves. This kind of assignment corresponds to variable initializations or overrides for program variables that were already declared earlier in the code, where the RHS is either some expression e.g., `x = (1 < 0)`, or a (external or C built-in) function, e.g., `x = sizeof(int)` for an earlier declared `int x`. Function calls of external and built-in C functions, that are not assignments, are handled as statement edges as well. For all the types of statement edges there is specific information of the statement like the corresponding expression or function call in case it is not an assignment, and the left- and right-hand side otherwise.
   A statement edge is one of the most common types of edge in a program and is the place where a big part of the logic of the analysis is implemented, because in most of cases here is where the information flows explicitly: the taint flows from the *RHS* to the *LHS* of an assignments (or declarations with non-null initializer).

4. **Function Call Edge**: This type of edge comes from program lines containing raw function call expressions like `foo(x,...)` or also assignments such as `y = foo(x,...)` for given program variables x, y and, in contrast with the *Function Call Statement Edge* explained above, a not-external and not-built-in function `foo()` that is defined outside the scope of the current function.
   This kind of edge contains information about the called function, e.g., the name of the function or the passed parameters if present. This and the next type of edge are particularly relevant for the handling of inter-procedural function calls in the analysis.

5. **Function Return Edge**: This edge connects the last node of a function and the location where the function was called. It is strongly related to a *function call edge*, because it is created at some point after a *function call edge*, no matter whether the called function has a return value or if it is a void function. This kind of edge includes the information about the inter-procedural flow, i.e., from which and to which function flows the returned value, and, when the called function has a return value, this edge allows the handling of it.

Listing 4 shows an example program in which one can clearly see the difference between the last two mentioned edges and a *return statement edge* which will be explained next. The lines 3 and 7 trigger both a function call edge and a function return edge while line 11 triggers a declaration, a statement and a return statement edge.

```
1  int main() {
2      int a = 2;
3      foo(a); // fct. call edge and fct. return edge
4  }
5
6  void foo(int p) {
7      p = taint(); // fct. call edge and fct. return edge
8  }
9
10 int taint() {
11     // declaration edge and statement edge
12     // due to internal CPAchecker temp vars
13     return __VERIFIER_nondet_int();  // and return statement edge
14 }
```

Listing 4: Example function call edge vs function return edge vs return statement edge

6. **Return Statement Edge**: This edge is triggered directly by a return statement in the code, and therefore, it handles the return value of a function. The attributes that are particular for this type of edge are the expression that is being returned, and an assignment, which is created internally and named `__retval__` (for *return value*) and has the purpose of resolving the returned expression and saving it into the internally generated *retval* variable for later handling outside of the function in which it was created.
   For example, in figure 4 the value of `__VERIFIER_nondet_int()` will be stored in `__retval__` and then the value of it will be saved in the variable `p`. The variable `__retval__` generated in a return statement edge works as a sort of bridge between function calls, and allows us to handle the taint propagation in inter-procedural calls.

It is important to note that one line of code can imply several types of edge. In general, two things can happen when the analysis processes an edge: either a new "good" state is generated, or a property violation is set for the current state. The specific handling of each of these edge types in our implementation of taint analysis is explained in detail in chapter 5.

The code in snippet 5 shows a static analysis of the edges that each line generates. Note that, to keep the comments short, only the name of the generated type of edge was written, but it does not mean that a line is an edge. A source code line and a CFA edge are closely related concepts, but they are distinct and cannot be used interchangeably. The comments that are written as "`<line of source code> //<edge type>`" must be read as "`<line of source code> // generates an <edge type>`".

Notes about listing 5:

- Code line 9 triggers two *assume edges*, one for `arg == 0` and one for `!(arg == 0)`.

- Code line 10 and 15 trigger two edges, one *function call edge* before processing the function's body of the function `taint()`, and one *function return edge* after the called function was processed. Note that the word "return" in the *function return edge* does not refer to the return value of a function, it refers to the edge that is "comming back" (returning) from the called function to its caller.

- Code line 12 triggers a statement edge, whose assignment RHS contains the evaluated value of `sizeof(int)`, in this case `4UL`. This does not only apply to the `sizeof` function, but to all built-in C functions, and is something that CPAchecker already delivers in the parsing of the given code, so the analysis does not have a way to handle parameters that were given to built-in C functions. This last point makes it difficult to track taint propagation for such function calls and its handling is explained in Chapter 5.

- Code lines 24 to 26, regarding the function `taint()`, if the return statement would have been directly `return __VERIFIER_nondet_int();`, or more generally, if the returned expression is not a declared variable, CPAchecker internally creates a variable `<Return type of the called function> __CPAchecker_TMP_<index>` in the scope of the called function, to store its return value. For this it creates an statement edge which assigns the returned value to the newly created variable: e.g., `int __CPAchecker_TMP_0 = __VERIFIER_nondet_int();`

```c
extern int __VERIFIER_nondet_int(); // declaration edge
extern int __VERIFIER_is_public(int variable, int booleanFlag);

int main(int arg) { // declaration edge
    int a = 2; // declaration edge
    a = 2 < arg; // statement edge
    int x; // declaration edge

    if (arg) { // two assume edges, (arg == 0) and (arg != 0)
        x = taint(); // function call edge and function return edge
    } else {
        x = sizeof(int); // statement edge
    }

    foo(&a); // function call edge and function return edge

    __VERIFIER_is_public(a + x, 0); // statement edge
}

void foo(int* p) { // declaration edge
    *p = 2 * (*p); // statement edge
}

int taint() { // declaration edge
    int a = __VERIFIER_nondet_int(); // declaration edge (for `int a
        ;`) and statement edge (for the assignment `a = ...`)
    return a; // return statement edge
}
```

Listing 5: Example C code snippet containing all types of edges that are handled in taint analysis

4.3 Parsing of C expressions

CPAchecker delivers a robust parsing of expression for the analyzed C programs to the implemented analysis. We handle some of them to model the taint analysis, but there are still some that could be added in a future extension of the analysis. A summary of the used expressions in our implementation follows:

1. **CExpressions**: Is the interface used to model further types of expressions. All expressions that we use in the analysis implement this interface.
2. **CLiteralExpression**: This represents literal values, for example, `(int) 5`, `(char) 'c'` or `(float) 1.2`.
3. **CIdExpressions**: This type of expression represents a single variable name, e.g., `x`, `y`, `var`, etc.
4. **CBinaryExpression**: This represents operations with the form `x ○ y` where `○` is a binary operator, and `x` and `y` can be CExpressions themselves, including further binary expressions. For example, the expression `a + b * c` is parsed as a binary expression by considering the usual binding arithmetic rules (e.g., multiplication binds stronger than the sum operator). With this, the term `y`

`:= b * c` becomes one term in the binary expression and as result the parsed binary operation is considered as **a + y** which is resolved by unfolding the results of the respective binary expressions.

5. **CCastExpression**: Contains information about the new data type and the casted CExpressions.
6. **CPointerExpression**: Used to handle occurrences of pointers, for example, `*p = 2` or `int *p = &x`.
7. **CUnaryExpression**: Used to handle occurrences of unary operations applied to a CExpressions, for example, `-foo` or `&var`.
8. **CArraySubscriptExpression**: Used to handle occurrences of arrays, for example, `a[1] = 3`.
9. **CFieldReference**: Used to handle occurrences of field references, for example, `d.field = x` or for pointer dereference: `p -> value = x`.

All this information is automatically parsed by CPAchecker and can be used afterwards to handle the different situations.

CHAPTER 5

Taint Analysis in CPAchecker

As explained in chapter 3, our taint analysis implementation is based on the formal definition from the paper Lazy Self Composition [7]. Although it must differ from it in some points, due to the context of CPAchecker.
An example of one deviation from the definition in the paper, that we must take for our implementation, are the CPAchecker C-built-in keywords `sizeof` and `__alignof__` or `_Alignof`. These functions are internally evaluated at the program parsing in CPAchecker before they are passed to the taint analysis, and for this reason the analysis cannot see the arguments that were passed to that function. Without this information, it is not possible for the analysis to track the taint flow in such calls. These functions can represent a source of sensitive information, but as long as we don't have a way to access the original passed variables, we must accept that for such cases the analysis will fail. Additionally, the behavior of the analysis for these cases, is not consistent with the most general overapproximation rule of taint analysis, that in every statement a tainted right-hand-side (RHS) should taint the respective left-hand-side (LHS). To make this situation clear, we see in figure 6, an example C program that uses the `sizeof` function with a tainted parameter `x`. If we strictly follow the definition of taint analysis, then the RHS of the expression in line 3 must taint its LHS, which does not happen in our implementation, due to the mentioned limitation.

```
1
2 int main(int sensitive_information) {
3     int a = sizeof(sensitive_information);
4     return a;
5 }
```

Listing 6: Example C program that returns the sizeof value of sensitive information.

5.1 Threat Model

In information flow analysis the *threat model* determines when the analysis considers that an attacker would be able to observe sensitive information based on the program execution. For defining our threat model, we first need to make clear that in contrast to the definition of the paper lazy self composition for security verification [7], we do not consider fixed sets of high and low-security variables. Instead, we consider the set X of all the program variables and two subsets T and U, which represent the set of tainted and the set of untainted variables, respectively, with $X = T \cup U$. The variables $x \in X$ are moved along the sets of tainted and untainted variables depending on its current *taint-state*, and therefore, in our implementation a low-security variable is the same as an untainted value and therefore, it can not happen that a program variable is both low-security and tainted as the same time.
A program variable being moved to the set of tainted variables becomes a high-security variable and is therefore equivalent to the combination of a variable with its corresponding taint flag from the formal definition (and the same applies for the set of untainted variables).
Now, let P be a program over the set of program variables X. For $x \in X$ we consider the predicate $Obs_x(X)$ over program variables X which will determine if x is *adversary-observable*. We use the simplest thread model approach suggested in the paper: "For each low variable $x \in L, Obs_x(X)$ holds at program completion". In our case we use U instead of L and it corresponds to a threat model where the adversary can observe the untainted variables after program completion. [7]
However, we consider a more general condition to determine property violations than only at program completion: we check the taint on each arbitrary moment of the program execution. A continuation of this work could be to implement the property violations totally based on the thread model, defining sinks that would follow the same logic as the current function that we use to check for property violations. For now, the main goal of this work is to make sure that the analysis behaves as we expect, and for which we need to observe and measure the behavior of the analysis during the program execution and not only at the end of it.

5.2 Taint Analysis State

To understand how property violations are handled in our taint analysis implementation, we will first look at the abstract states. *Taint analysis states* are modeled in the class `TaintAnalysisState.java`. It maintains the sets T of tainted and U of untainted variables, sets of *predecessor*, *successor* and *sibling* states of the current state, a boolean flag *violatesProperty* which indicates when the state violates the information flow property, a boolean flag `isPathStart` which indicates whether the current state is the start of a split path after a control structure condition, and a list `nonTrivialPathStartStates` of all the path start states that precede the current state. The predecessor states are used for the `isLessOrEqual` check that compares two states and determines whether one state is covered by another. The successors are only used inside the constructor for determining whether a new

generated state is successor of an state that has two successors, and therefore, to determine whether the new generated state has a sibling state and is a path start state. The list of `nonTrivialPathStartStates` is used to check efficiently whether a state precedes another state. To implement the path-sensitivity of the analysis the predicate analysis CPA should have been used. However, I realized this too late in the developing of the module and could not figure out in time how to integrate the value tracking into the taint analysis using an existing module dedicated to this. That is why I had to add one field related to value tracking that conceptually should not be in the taint analysis module and should be moved from there to a different CPA. This field is a map `evaluatedValues` that maps each program variable to its value in the current state. Keeping a future refactoring of this in mind, we separate the taint tracking so much as possible from the value tracking, the `evaluatedValues` does not keep track of the taint, and the `tainted` and `untainted` lists do not keep track of the values. The `evaluatedValues` map is used in the `TaintAnalysisTransferRelation` class to keep track of reachable and unreachable branches, and in the `TaintAnalysisState` class for the state merge process. The class `TaintAnalysisTransferRelation.java` is responsible, among other things, of the generation of new states for the CPA Algorithm.

5.3 Merge of states

As part of the state space exploration, the CPAalgorithm needs to check whether explored states are covered by others by comparing them. This is where the `join` method comes into play. The join method of the taint analysis module, compares two states; `this` and `other` and checks first whether `this` state is `lessOrEqual` than the `other` state, if not it checks whether the `other` state is `lessOrEqual` than `this` state. If the previous checks failed, then it proceeds to merge (join) them. Without value tracking, the merge process should create a new state that carries all the tainted variables that were found in the joined states, and only the untainted variables that were untainted in both states. With value tracking this is a little more complex: The join of the taint status of the variables remains the same, but additionally the analysis needs to check whether the variables contain the same mapped values. If variables contain the same mapped values the join is made as normal, only one state is generated, containing the combined taint status of the parent states. If the parent/joined states do not contain the same values, the join method will create a new false state that contains the combined taint status of the parents, but maps each variable to two (or more) values. This new state is a sort of state container that will carry the values of the parent states with the usual combined taint status. The states container will then be unpacked in the transfer relation class, where the parent states are reconstructed, with single value mapping and updated taint status. In this way, for example, when a states container that contains n states enters the transfer relation class, will generate n states with single value mapping and all n states will have the same updated taint. All the unpacked states will then be checked against the same edge for property violations in the transfer relation, each state containing a different `evaluatedValues` map.

5.4 Information Flow Violation

We are interested in checking whether expected taint status holds at certain points of the program execution rather than knowing the status only at the end of it. Therefore, our definition of a *bad-state* does not only consider the threat model, but also whether the expected taint status holds at certain points of the program execution. Therefore, we generalize the bad state definition of the paper. The result is slightly different: Let $t\colon X \to \mathbb{B}$ be a function that takes a program variable and returns its *current taint-status*; $true$ when it's tainted and $false$ otherwise, and let $e\colon X \to \mathbb{B}$ a function that in the same way returns the *expected taint-status* of a program variable.

We define a *Bad-State* as $Bad(X) := \bigvee_{x \in X}(t(x) \neq e(x))$, a state over all the program variables X in which the expected public/taint status of some checked variable was different from its current taint status, and not necessarily a state in which information is leaked. This is not only useful to identify information leaks, but also to evaluate the performance of the analysis, and identifying inconsistencies fast, e.g., the analysis reporting a program variable as tainted in a safe program and untainted in an unsafe program. The program in figure 7 shows an example of a situation in which our analysis will return an information flow violation, without any leak of information being present, but with the expected taint not matching the current taint status.

```
1
2 int main() {
3     int a = 3;
4     __VERIFIER_is_public(a, 0);
5 }
```

Listing 7: Example C program fails without finding a tainted variable.

At this point it is important to note that we do not use *SINKS* in our taint analysis. The original prototype included a list of SINKS, e.g., the function `printf();`. We replaced this list of SINKS with one check-function `__VERIFIER_is_public(<data_type> var, <int> expected_public_status);`. There are two reasons for this. The first one is that once the analysis is implemented, the SINKS could be implemented again, e.g., by passing the SINKS to the analysis via configuration file. For the benchmarking of the implementation, it is easier to maintain only one unifying function that allows us to model its behavior. The second reason is that this unified function allows us to explicitly check if a program variable is tainted or if it is public. This is important because it allows us to evaluate the analysis with safe and unsafe versions of the benchmarks covering scenarios in which an incomplete implementation of the analysis could report inconsistent analysis results. Such cases would be recognized immediately by the publicity-check.

5.4.1 __VERIFIER_is_public function

The `__VERIFIER_is_public(<CExpression>, <boolean_flag>)` function receives two parameters, the first can be any type of C-expression (single variable name, binary expression, etc.), and the second argument represents the **expected** public/taint status of the given expression in the first parameter. It is the only function in the code base that is meant to check the public/taint-status of a given expression, and it is therefore the one responsible for identifying when the information-flow property is violated and marking the current program state accordingly. Its use can be seen in figure 7, where the variable `a` is expected to be tainted (i.e., not-public). It returns 1 when the assertion holds and 0 otherwise.
This is a critical part of the analysis, since it directly influences the state generation and therefore the state space exploration executed by CPAchecker. Note that aside from single variable name expressions this function supports all kind of expressions.

5.5 Path sensitivity

A path-sensitive taint analysis does not explore unreachable branches. The path sensitivity is not mandatory for information-flow or taint analysis, but it improves the precision of it, at the cost of complexity in the implementation. Taint analysis is an overapproximation thus it causes false positives [lazy self composition paper]. Although, a not-path sensitive taint analysis reports more *false positives* than a path sensitive analysis, due to the over-tainting in unreachable branches. In the language of taint analysis, a *false positive* arises when the analysis reports a security leak where there isn't, and in the same way, a *false negative* reports no information disclosure when it should. The reason for the increased number of *false positives* when exploring unreachable branches is evident: unreachable taint-propagation could be triggered due to the more general overapproximation.
Figure 8 shows an example C program that contains an unreachable statement in line 7 and assumes that a function `taint();` is a source of sensitive information. A not-path-sensitive taint analysis will consider the statement `b = taint()` as a valid information flow from the function `taint()` to the variable `b` thus reporting a security leak when there is not. In contrast, a path sensitive analysis will not explore the unreachable statement.

```
1
2  int main() {
3      int a = 2;
4      int b;
5
6      if (a < 0) {
7          b = taint();
8      } else {
9          b = a;
10     }
11
12     return b;
13 }
```

Listing 8: Example C program containing an unreachable explicit taint-flow

5.6 Taint policy

5.6.1 Taint generation

We need to define what a source of sensitive information is, the so called *Sources*. The class `TaintAnalysisTransferRelation.java` maintains a list called `SOURCES` for this goal. The list of sources can be expanded, but for the sake of implementing the analysis and benchmarking it, we consider for now two different sources of high-security values:

- The function `__VERIFIER_nondet_<data_type>();`, where `<data_type>` can be of type *int*, *char*, *float*, or *double*. It does not actually return any value. The call `int x = __VERIFIER_nondet_int();` stores `null` in x, but semantically assigns to x the status tainted.

- For safety, we adopt a conservative approach for arguments passed to the main function. These arguments are also considered to be tainted. E.g., in figure 9 the variable `arg` is considered to be tainted from the beginning. This conservative approach is a source of *false positives* since not every argument passed to the main function is a real source of sensitive information. A less conservative handling of this situation would increase the precision of the analysis.

```
1
2  int main(int arg) {
3      //...
4  }
```

Listing 9: Argument passed to the main function is considered to be tainted

5.6.2 Taint propagation

As explained in Chapter 3, the basis for the behavior of our analysis is the formal definition of taint analysis in the paper lazy self composition for software verification, and this applies for the modeling of the taint propagation in the transfer relation class. The definition for our implementation remains unchanged, since our analysis is path-sensitive and considers both, explicit and implicit information flow. These concepts also build the basis for the *ground truth* of our benchmark set. The documentation of the external benchmarks that we used follow, in general, the same rules for taint propagation.

We recall that in the definition presented in Chapter 3 $x'_t \in X'_t$ represents the updated taint status of the program variable $x \in X$. In some of the benchmark programs, we use $t(x) = x'_t$ to indicate the taint status of a program variable $x \in X$. The formula above can then be read as

$$\Theta(\varphi) = \begin{cases} \text{false} & \text{if } \varphi \cap X = \emptyset \\ \bigvee\limits_{x \in \varphi} t(x) & \text{otherwise} \end{cases}$$

The use of $t(x)$ instead of x'_t is just for simplicity in the code comments, and does not imply any semantical change.

Listing 10 shows the taint status of the program variables $x, arg, sum \in X$ through the program through comments in the code. This kind of comment is present in several benchmark programs as documentation of the *ground truth*, which represents the expected taint values of program variables at certain points of the program execution. For simplicity in the program comments we use the symbol $+$ instead of $\vee$, `T` for tainted (`true`) and `U` for untainted (`false`).

```
1
2  int main(int arg) { // t(arg) = T
3      int x;
4      if (arg) {
5          x = 2; // t(x) = U
6      } else {
7          x = taint(); // t(x) = T
8      }
9      // t(x) = U + T = T
10     int sum = arg + x; // t(sum) = t(arg) + t(x) = T + T = T
11     return sum;
12 }
```

Listing 10: Comments to clarify the taint propagation

In the following we list the policy for the taint propagation regarding certain structures.

5.6.2.1 If-Statements and loops

When analyzing a control structure, there are generally two options: the condition of the loop can be evaluated or it can not. This is explicitly checked in our analysis in the

handling of an assume edge, as explained in the past section. Our analysis generates a new state in case the condition holds or in case it can not be evaluated, and no new state otherwise. The policy for taint propagation inside if-statements is at this point fully covered by the explanations about explicit and implicit information flow, and the fact that our analysis is path sensitive. For loops hold the same behavior regarding the conditions and the path sensitivity. However, a particularly interesting case for loops is the one shown in figure 11. In the snippet we can observe an untainted variable `a` that controls the loop condition and a variable `b` that gets increased inside the loop. But then `a` gets tainted inside the loop, and its value is replaced with an non deterministic value. Now, after leaving the iteration, the condition is checked again, but the analysis determines that it can no longer evaluate the condition. The next step is to go on with the program execution going in two separate ways, one outside the loop and one inside. Here we encounter two problems: we need to be able to join the path that goes inside the loops and the one that does goes outside of it. The second problem is that, given that the loop condition now cannot be evaluated, the stop condition will never be met, and therefore the analysis needs to know when to stop exploring inside the loop, otherwise it will fall in an infinite execution inside the loop. The solution for the first problem is given by CPAchecker, it automatically triggers the join of the states that are coming together after a branching. The solution to the second problem must be implemented in the analysis, because, given the value tracking, the analysis will constantly increase the variable `b` making it imposible to merge two states that despite the same taint, differ in the value `b`. For addressing this, we use the control structure objects (`loopStructure`). With help of these objects we can determine whether a variable is being increased inside a loop that is controlled by a variable that can not be evaluated. If that is the case, the variable will not be increased, allowing the merge of the duplicated states, which will trigger that the exploration of that branch terminates. In the first loop-iteration post taint of `a`, the variable `b` receives one more assignment `b++;`. Since this happens inside a loop controlled by a now tainted variable, `b` becomes implicitly tainted, but the value will not be updated, because the condition is not-evaluable due to a non-deterministic `a`. Since the taint and the values don't change in the next iteration, the future states of that path are considered as covered, terminating with the exploration of that path, leaving the exploration of the loop. The join of the taint of b will then recognize `b` as tainted. Summarized: if there is one path in which a tainted `b` reaches a sensitive program location, then `b` must be considered as tainted in that location.

```
1  int main() {
2      int a = 2;
3      int b = 0;
4
5      while (a > 0) {
6          b++;
7          a = __VERIFIER_nondet_int();
8      }
9
10     __VERIFIER_is_public(b, 0); // t(b) = U + T = T
11 }
```

Listing 11: Problem generated by a null condition

This applies to for-loops as well. The analysis does not support this level of handling of do-while loops, because I overlooked this loop type for a long time. The implementation is very similar to a while or for-loop, but I did not have the time to implement it. An approach for addressing this is presented in the section about future work 7.
Note that the ternary operator also support the exact same functionality as an if-statement, including the implicit taint propagation.

5.6.2.2 Arrays and field members

Taint of arrays is supported and obeys the normal rule of RHS taints the LHS. And can also get tainted by implicit taint. However, the sanitization of an array is slightly different, since an untainted LHS does not sanitize automatically the RHS. Only after every element was sanitized the array can be considered as untainted. The logic for the implementation for field members is exactly the same as the one for arrays, but instead of elements it considers the fields as tainted.

5.6.2.3 Pointers

With help of the value tracking we can taint or untaint pointer and the pointed memory addressed directly. In this case we do not keep track of the actual memory address, but of the expression that represents that memory address. The taint policy followed here is that by tainting a pointer, all the pointed memory addresses must become tainted, and the same for sanitization. In that same way, is a variable gets tainted or sanitized, the same changes must be applied to the corresponding pointers if there are. This works, because the `evaluatedValues` maps CExpressions to CExpressions, therefore, a mapping, for example, from `p` to `&y` is possible. Since the `evaluatedValues` keeps track of all the existing program variables, it will always find the correct mapping to the right CExpression.

5.6.2.4 Interprocedural function calls

The analysis supports taint propagation through function calls. For this there is no specific restriction. In these cases, the goal is as usual to keep track of influences between variables among function calls, e.g., if a function returns a tainted value, and the return value is assigned to a variable `x`, then x will become tainted. If a void function makes some modification in the memory address of a variable that is defined in the scope of another function, the corresponding variable will become tainted as well. Taint propagation for global variables is also supported.
In cases in which we are not certain of the taint status of a variable, the analysis will behave conservatively by considering these as transitions to security leaks.

5.6.3 Taint Removal - Sanitization

The sanitization of variables is an important part of the taint analysis, because in real world programs there are scenarios where sensitive inputs can become sanitized and

therefore not be high security values any more. Sanitization refers to the application of specific checks, transformations, or validations to remove harmful or unexpected content from an input. For instance, validating and escaping special characters in user-provided text fields to prevent SQL injection or XSS (cross-site scripting) attacks is a common sanitization method. Just like the *Sources*, the sanitization methods are specific to different program contexts. To generalize the sanitization process, our analysis uses the function `__VERIFIER_set_public(var, bool)`, where `var` is the variable to be modified as a `CIdExpression`, and `bool` $\in 0, 1$ determines whether `var` will be considered as tainted (`var` $\in T$) or as sanitized (`var` $\in U$), 0 for tainted and 1 public (untainted).

This is useful on one hand for emulating a sanitize-function, and on the other hand as a general tainting-function. After being called, the given variable will be flagged as the boolean parameter says, and then the state generation step will move the variable from one list (tainted/untainted) to the other if necessary. This public-status change is effective in the most of cases, but not always. The impact of using this function is restricted by the tainting logic followed by the analysis, therefore, there are some cases in which this call does not have any impact on the public-status of a variable. Figure 12 shows a code example of a sanitization attempt over single elements of a tainted array. In our implemented analysis this program will not throw any information-flow violation, meaning that the attempt of sanitization fails. This behavior is intentional and obeys the logic that when at least one element of the array is tainted, then all the other elements must also remain tainted. However, an array can only be sanitized by sanitizing the array itself with `__VERIFIER_set_public()` or by sanitizing every single element in the array. Additionally, the analysis will internally recognize that `d[1]` was sanitized, but it still will be reported as tainted, for being an element of a tainted array. A point of discussion can be set here: for precision the analysis could report the singe untainted variables as untainted no matter if they are part of a tainted array. For this I decided to adopt the most conservative approach to only report the variable as untainted, when the array is untainted.

```
1
2  int main() {
3      int x = 0;
4      int y = __VERIFIER_nondet_int(); // t(y) = T
5      int z = y;
6
7      // taint flows from y to d
8      int d[2] = {x, y, z};
9
10     __VERIFIER_set_public(d[1], 1);
11
12     // The array is expected to remain tainted, because it still
           contains a tainted element
13     __VERIFIER_is_public(d, 0);
14
15     // The array elements are also expected to be reported as tainted
16     for (int i = 0; i < sizeof(d) / sizeof(d[0]); i++) {
17         __VERIFIER_is_public(d[i], 0);
18     }
19 }
```

Listing 12: Example code of tainted array, with an attempt to sanitize single elements of the array

There are other, more specific ways to taint or untaint a variable such as simply override it with an untainted RHS (in the most cases), but we keep this function as a general purpose taint/untaint-function, e.g., for modeling some of the benchmarks that I created to test specific program scenarios for the implementation, or to "translate" or emulate the behavior of some code parts contained in extern benchmarks.

An important consideration is that the subtraction or arithmetic manipulation of sensitive variables does not automatically result in sanitization; explicit operations or reassignments are required to remove or override taint status.
For instance, consider the following illustrative scenario in figure 13.

```
1  int main() {
2      int x = __VERIFIER_nondet_int(); // sensitive data
3      int y = 5;
4      y = y + x; // taint flows from x (RHS) to y (LHS)
5      int z = y - x; // subtracting x won't sanitize x - y
6      __VERIFIER_is_public(z, 1); // asserts z is public will fail
7  }
```

Listing 13: Example benchmark scenario demonstrating taint propagation

In this example, despite arithmetic operations that might intuitively suggest sanitization, our implementation correctly identifies the variable z as tainted. This behavior follows the general taint propagation rule (overapproximation), that a tainted variable on the *RHS* will taint the *LHS* of the statement. Hence, the assertion posed by `__VERIFIER_is_public(z, 1)` explicitly violates the expected taint-property and is correctly flagged by our tool as unsafe. In such case, we encounter a *false positive* verdict, since the final value of `z` is `y` which is a constant value 5 and therefore

not tainted. However, the actual logic that the analysis follows can be seen in figure 14.

```
1 int main() {
2     int x = __VERIFIER_nondet_int(); // t(x) = T
3     int y = 5; // t(y) = U
4     y = y + x; // t(y) = t(y) + t(x) = U + T = T
5     int z = y - x; // t(z) = t(y) + t(x) = T + T = T
6     __VERIFIER_is_public(z, 1);
7 }
```

Listing 14: Example benchmark scenario demonstrating taint propagation

5.7 Taint Analysis Transfer Relation

Now that we have reviewed the most relevant characteristics of taint analysis and the related aspects of CPAchecker, in this section we are going to explain the handling of important structures of C programs that are implemented in our taint analysis. The handling of all the different edges that a program contains is made in the class `TaintAnalysisTransferRelation`. The transfer relation class contains as fields a list of `SOURCES` that was described in the section about the taint generation 5.6.1 and, additionally, for the tracking of implicit information flow in control structures such as loops of if-statements the transfer relation receives the `loopStructure` and the `astCFARelation` when created, so that it can access the information about the control structures and propagate the implicit taint when necessary.
The entry point of the transfer relation class is the method `getAbstractSuccessorForEdge` that receives as input an abstract state, a precision, and a cfa edge and returns a collection of new states. The precision is not used in our analysis. The abstract state carries the taint and the values generated in its predecessor state, and the cfa edge represents a line of code or another edge that connects two program nodes.
Before the handling of the received edges the transfer relation makes three important steps: it calculates what type of edge it received, then checks whether the current edge is leaving a control structure, and then it attempts to extract states from the received state, in case it is a states container. The reason for checking if the edge is leaving a control structure is to force a join if necessary. This is, again, a consequence of the value tracking that was implemented as last resort inside the taint analysis module: The value tracking influenced the join dynamics, and the join dynamics influences the state generation. This was not an issue in the most cases, but particularly for one benchmark case in which an if-statement was nested inside a while loop, CPAchecker was not joining the two execution paths, resulting in a property violation not being eliminated. To compensate this, the transfer relation checks this and in case there are property violations in other execution paths that were found before the end of the control structure, it forces the join of them so that the property violation can be checked again. Therefore, the value tracking is definitely a part of the current analysis design that should be separated from the taint analysis module

to a different CPA. The extraction of states of a states container is also connected to the same issue, and the reason for it was explained in section 5.2.

After these checks the method will handle either the incoming edge, or a list of extracted states. In any case, the next step is to handle the edges depending on their edge type, and either generate, propagate (implicit and explicit taint), or remove taint. The taint propagation is made in each method that handles the different edges by populating the sets `killedVars` and `generatedVars`, following the rules described in 5.6.2. For the variable-values mapping the map `values` is populated. The sets and map are then passed to the `generateNewState`, that has the responsibility to create a new state based on the new found taint. Note that the taint status of all variables could remain unchanged after the handling of the edge, in which case CPAchecker will eventually attempt to trigger the merge of these states.

In the following we will describe how the transfer relation class handles the edges mentioned in section 4.2 and how they handle the types of expressions mentioned in section 4.3.

5.7.1 Assume Edge

Assume edges are handled in the method `handleAssumption`. This method has two responsibilities: to evaluate the given condition and to generate a new state in case the condition holds. For the evaluation of the given condition it retrieves the `CBinaryExpression` contained in the assume edge and uses the values that the state carries in the `evaluatedValues`.

We present the example in figure 15 as motivation for the importance of a path sensitive analysis. In the given program a path-sensitive analysis produces a different result than a non path-sensitive analysis. We can see that the statement `__VERIFIER_set_public(a, 1)` is reachable and, therefore, the variable a is sanitized. However, a non-path-sensitive taint analysis will explore both branches of the program: `b == 1` and `!(b == 1)`. It will also ignore the line `__VERIFIER_set_public(a, 1)` in one program path, which will lead to the taint check `__VERIFIER_is_public(a, 1)` to set a property violation in that analyzed program state.

```
1  int main() {
2      int a = __VERIFIER_nondet_int(); // t(a) = T
3      int b = 1; // t(b) = U
4      if (b == 1) {
5          __VERIFIER_set_public(a, 1); t(a) = U
6      }
7      // path sensitive: branch  t(a) = U = U
8      // non-path-sensitive: t(a) = T + U = T
9      __VERIFIER_is_public(a, 1);
10 }
```

Listing 15: branches

5.7.2 Declaration edge

The method `handleDeclarationEdge` handles declarations of type `CVariableDeclaration` and `CFunctionDeclaration`. The first type covers all the declarations inside a function body, and the second one covers the variables passed to functions as parameters, for example it would handle the declaration of the function `int foo(int var)`. If the initializer of the expression is null it just adds the LHS as an untainted variable with value null, and in case the initializer is not null it must check what type of initializer is it. In our modul the initializers of type `CInitializerExpression` and `CInitializerList` were the only ones that needed attention. However, in case another type of initializer would need handling, the analysis would log it, but not handle it. In case the initializer is an `initializerList`, it would correspond to a statement like `int x[3] = {1,2,3}`, which is currently only supported for arrays. This method does not handle implicit information flow, because variables that are declared inside a loop do not survive outside the scope of its function. This can be an interesting discussion point and maybe source for improvements.

5.7.3 Statement Edge

Like stated before, because of the structure of programs, this kind of edge is one of the most used and the one where the most of different cases must be handled. The method `handleStatementEdge` is responsible for this, and covers the handling of the four special edge types mentioned in item 3. For both assignment types, the `CExpressionAssignmentStatement` and the `CFunctionCallAssignmentStatement`, the analysis extracts the LHS and the RHS and checks whether the RHS is tainted or if there is implicit taint flow, in any of these cases the taint flows into the LHS. In the case of a `CFunctionCallStatement` the statement does not contain an assignment, and therefore there is no RHS to check against taint. However, the functions `__VERIFIER_set_public` and `__VERIFIER_is_public` are CFunctionCallStatements. The first function changes the taint status of the entered first parameter if necessary, and the second function compares the current taint vs the expected taint, and in case they do not match, it generates a information-flow violation for the current state. The method `handleStatementEdge` handles all the types mentioned in 4.2.

5.7.4 Function Call Edge

This edge is handled in the method `handleFunctionCallEdge` which first checks whether the called function has any arguments. If not it just calls the `generateNewStates` with empty sets of tainted and untainted. If the function called has arguments on the caller line, it connects the passed arguments to the function arguments on the called function and taints it in case the argument was tainted in the caller. To illustrate this, see the snippet in figure 16, where in line 5 the function `foo` is called with an untainted argument `&a`. This will result in the pointer `int* p` on line 247 being untainted at the beginning of the function. Note

that the analysis will differenciate between the pointer `p` defined in line 3 and the one inside the `foo` function, due to the qualified name of the variables (`main:p` vs `foo:p`) and the robust parsing of CPAchecker. Therefore, they will be handled as different variables, even if they have the same name.

```
1  int main() {
2      int a = 2;
3      int *p;
4
5      foo(&a);
6      // rest of the program...
7  }
8
9  void foo(int* p) {
10     *p = taint();
11 }
```

Listing 16: Function call edge

Here there is room for improvements, since this part of the analysis still does not consider implicit taint flow. Additionally, it does not differentiate when the function is being called, for example, from inside a loop controlled by a condition that can not be evaluated. This special case is very important regarding state generation and termination of the analysis: if the conditions mentioned are met, the analysis will explore both branches and if it does not recognizes this it will keep updating the values inside the loop, causing in many cases that the analysis times out, due to a non terminating generation of new different states. This is actually the reason for the timeout of some benchmarks as we will see in the benchmark section. This scenario is handled in the `handleStatementEdge` and other edges, but not yet here.

5.7.5 Function Return Edge

This edge is the bridge between caller and called method, and is handled in the method `handleFunctionReturnEdge`. It determines whether the function call is not void, i.e., it has a return statement and if so it retrieves it from the edge information. Then, it checks whether the return value is tainted and, if additionally the related expression is an assignment, it propagates the taint to the LHS. LHSs of type `CIdExpression` and of type `CPointerExpression` are supported here, and the benchmark set did not require more than that. However, more cases should be considered, such as field access, arrays, etc. This part of the analysis is, in that aspect, still incomplete and would also profit from support for implicit information flow recognition.

5.7.6 Return Statement Edge

This edge checks whether the return value is tainted and if so it taints a temporary value. Additionally, it checks whether the current function is the main function, and whether it is returning a tainted value. This is the only part of the code aside of the check `__VERIFIER_is_public` that can set a property violation. This is actually not used in the benchmark set, but it shows how sinks could be handled in programs that use real functions as sinks.

5.7.7 Blank Edge

The blank edge is not really handled by the taint analysis, although there is a method that triggers the generation of a new state for this edge. This is relevant because without the generation of states for blank edges, the analysis would terminate before exploring the whole state space. A similar case is the handling of `CallToReturnEdge` that is present as part of the first prototype, but I have never seen a benchmark that uses it. Because of this I did not give priority to investigate more about this edge. Its functionallity is limited to generate a new state with the same values that the current state carries, just like the blank edge does.

5.7.8 Explicit vs Implicit information flow check

The explicit information flow does not have an special method in the transfer relation class. The approach for this is simple: in an assignment, the RHS taints the LHS. So we oft check first whether the RHS is tainted and then taint the LHS. The case of the implicit flow is more complex and requires the use of the structures mentioned at the beginning of this section to the transfer relation class 5.7. The loop structure and the astCfaRelation toghether with the location of the edges provide a clear way to know whether an statement is placed inside a loop, or any other control structure in the program. Additionally, one can check whether the corresponding loop is a nested loop, or if it is nested inside an if-statement, and other further imaginable cases. All this information is necessary to determine implicit taint information, as explained in Chapter 3. To check this, I created the two checks: `isInControlStructure(pCfaEdge)` and `isStatementControlledByTaintedVars(pCfaEdge, taintedVariables)` that use the mentioned fields to make the check and are present always when we want to determine whether implicit taint is taking place for the current edge.

CHAPTER 6

Benchmarking Taint Analysis

6.1 Benchmarking Set-Up

For the benchmarking we used CPAchecker version 4.0 and BenchExec version 3.30, and the BenchCloud to submit the runs in the cluster of the SoSy-Lab of the LMU-München. The benchmarks were executed in an Apollon machine with CPU Intel Xeon E3-1230 v5 @ 3.40 GHz, 8 cores, frequency: 3800 MHz, RAM: 33471 MB, using OS Linux 6.8.0-78-generic. The limits for the benchmark definition were: timelimit: 900 s, memlimit: 15000 MB, CPU core limit: 2.

6.2 Defining the benchmark set

The benchmark set has a total of 233 benchmarks from which several are the safe and unsafe variants of the same program, i.e., around 100 different programs. The benchmark set was collected from three different sources. The first source contains 46 benchmark programs that I wrote for testing the functionality of the implemented analysis during the development of the taint analysis module. Each program has at least one unsafe version, in total this makes 139 benchmarks that I called *core-modeling*. The second source is a set of 68 benchmarks from an open repository called *dceara*[1], some of them containing a safe and unsafe version. I choose this benchmark set because of its simple structure, but being slightly more challenging than the initial benchmarks, I saw in it a good suit for stressing the current state of the implementation to a point that helps me discover improvement areas. Additionally, it has some ground-truth documentation that uses almost the same taint policy as our analysis. The third source of 14 benchmarks is the one cited in the paper lazy self composition for security verification [2], the set is called `ifc-bench`.

[1] `https://github.com/dceara/tanalysis/tree/master/tanalysis/tests/func_tests`

[2] `http://www.cs.princeton.edu/~aartig/benchmarks/ifc_bench.zip`

The reason for selecting these benchmarks is because that paper is where I took the most foundations for building the behavior of the taint analysis in this work from. Additionally, the complexity of these benchmarks is superior than the others sets of benchmarks, and therefore it is a good candidate for stress-testing the implementation at the next level. With complexity here, I mean that the IFC-bench benchmark set combines several different structures and programming techniques, while the locally created and the ones from the repository dceara, are more tailored to test specific structures. The ground-truth is either well documented or can be easily inferred by the comments and naming of the benchmarks. In the benchmarks of the IFC-bench set, a program is safe when at the end of the program execution a checked variable is expected to be untainted, and unsafe otherwise. For testing the implementation with the collected extern benchmarks, I had to "translate" the programs to the language of our taint analysis modul, i.e., use the `__VERIFIER_nondet_<type>`, `__VERIFIER_set_public()` and `__VERIFIER_is_public()` instead of the checks that the benchmarks bring with them. A simple example of such a translation was to replace the verifier assertion `__VERIFIER_set_secret(1, var)` with our assertion `__VERIFIER_set_public(var, 0)`, in the ifc-bench set.
The benchmarking consisted of two runs: the first run considering all the benchmark sets together, and the second run where the benchmarks of the core-modeling and the ones from the dceara repository were classified in seven categories: arrays, cast, control flow structures, function calls, general functionality, pointers and struct-and-member-access.

The approximate coverage of the categories is as follows:

- 36% control flow structures
- 28% general functionality
- 9% arrays
- 9% pointers
- 5% function calls
- 2% casts
- 2% struct and member access
- 6% ifc-benchmarks

I could not include the categories of IFC-bench ad general functionality in any of the other mentioned categories because they don’t have a focus in testing a certain type of structure:

6.3 Results

6.3.1 Benchmark Run - All Benchmarks

The benchmark run of all benchmarks took a total of 18,5 minutes, with the benchmarks of the IFC-bench being the most ressource demanding. From a total of 233 programs, 216 were verified correctly, and 8 returned a wrong verdict. 3 incorrect true and 5 incorrect false. From the 233 benchmarks 9 timed out. Table n shows the summarized information.

Category	Status	CPU Time (s)	Wall Time (s)	Memory (MB)
All Results	233	1110	607	27800
Summary Measurements	-	-	181	-
Correct Results	216	972	531	24700
Correct True	81	372	203	9290
Correct False	135	601	328	15400
Incorrect Results	8	36.9	20.2	920
Incorrect True	3	13.2	7.23	344
Incorrect False	5	23.8	13.0	576

Table 6.1: Benchmarking results for taint analysis run all programs.

Correct Results

The categories in which the analysis had a perfect score were arrays (22/22), pointers (22/22) and member access (5/5) with 0 incorrect results each. Notably good performance can be observed for control structures (83/86) and function calls (11/12). In the category general functionality I left all benchmarks that I could not assign to a clear category. Despite in that category the analysis still is good (63 correct from 67), we can only affirm that our analysis shows a good basic functionality, since the benchmarks in this set are one of the most simple. The worst performance (3 correct from 5) we find for cast operations and for the benchmarks of the IFC-bench. That is expected, since cast support was not implemented and the IFC-bench benchmarks are the most complex, mixing a lot of different structures and operations in each benchmark. For the benchmarks of the IFC-bench the analysis had a score of 8 corrects from 14 and the remaining 6 timed out.

Incorrect Results

In following we analyze potential reasons for the failures. The reasons for the cast programs was already mentioned above: no support. The failure of the benchmark `trippleWhileLoopSafe` is due to an early termination of the analysis, before the found property violation can be fixed from the other execution path were no property violation is found, but where a bigger taint status is carried in the

state. This early termination is probably related to the issue of the value tracking mentioned in 5.2.

The programs **sizeof_allignof.c** and `taintBy*LogicalOperation*.c` fail due to the issue commented at the beginning of chapter 5. This behavior is deep integrated in CPAchecker and there is not much that the taint analysis modul itself can do to fix it. One option could be the string parsing of the raw statement where the lost variable was used. But string parsing is not a robust solution and should be avoided.

The programs `simpleBufferExample*.c` need the handling of the assignment `buf[n] = taint();` with `buf` being a pointer and `n` a constant. Assume that `buf` points to an arbitrary memory address `&x` of an int `x`. Then the expression `buf[n]` is equivalent to `&x + n`, which would mean to taint `&x + n`. This kind of tainting of a neighbor memory address is not yet supported by the analysis.

Timed-Out

The programs `simpleDoWhile*.c` fail to reach the line of the check because, as explained in subsection 5.6.2.1 it simply lacks of support. However, a clear solution for this is presented in the next chapter to conclusions and future work.
The rest of the time-outs correspond to the family of the IFC-bench set. Given the size of these benchmarks and that these cases were integrated at the end of this work, I could not debug the non-termination problem in time, but given the amount of loops and inter-procedural calls mixed, my guess is that somehow the states generation explodes at some point.

6.4 Observations about the categories

From the CPU time of the results that returned a correct verdict, we can corroborate the prior assumption of resource demands:

- **IFC-Bench Repository Benchmarks:** As expected, the run definition ifc-bench proves to be the most resource-intensive set. Its programs, such as `mod_add_-4096.c` and `pwdcheck_unsafe16.c`, have a steep CPU time growth. When analyzing these benchmarks, the runtime increases significantly even for a small number of benchmarks that return true, showing significant complexity and computational cost for these programs compared to the rest of the sets.
- **Non-control-strucure Benchmarks:** The sets of benchmarks without control structures shows the least resource demand, which aligns with expectations. These are fundamental cases with a simpler structure, allowing CPAchecker to process them efficiently.

The most demanding benchmark sets are the ifc-bench set and the control-flow-structures set. Some of the most demanding programs are:

- (from `ifc-bench`) `mod_add_4096.c` exhibits the sharpest increase in CPU time due to high computational complexity from large data structures and repetitive operations such as addition of big integers.
- (from `ifc-bench`) `pwdcheck_unsafe16.c` shows complexity due to character-level secrets comparison, which involves nested loops and branching.
- (from `dceara_set`) `tripplewhileloop_unsafe` and `innerwhileloop_safe` show steep computation times caused by deep nested loops that track taints over repeated iterations. Additionally, for this program the analysis succeeded in proving unsafety. However, for its counterpart, `tripplewhileloop_safe`, it failed to prove safety.

Performance Bottlenecks:

- Benchmarks with nested loops, large data structures, or implicit taint propagation mechanisms. The implementation spends significant time traversing and validating these intricate flows.
- The plots suggest that optimizations in handling nested control flows, such as loop unrolling or taint propagation minimization, may help reduce CPU time for complex sets like `ifc-bench` and `control_flow_structures`.

That loops are more ressource intensive than other structures such as if-statements or simple assignments is not a surprise, however, the additional features related to explicit value tracking that were implemented in TaintAnalysisState are probably a factor that plays an important role in the performance of the analysis, due to extra join operations and packing and unpacking states containers.

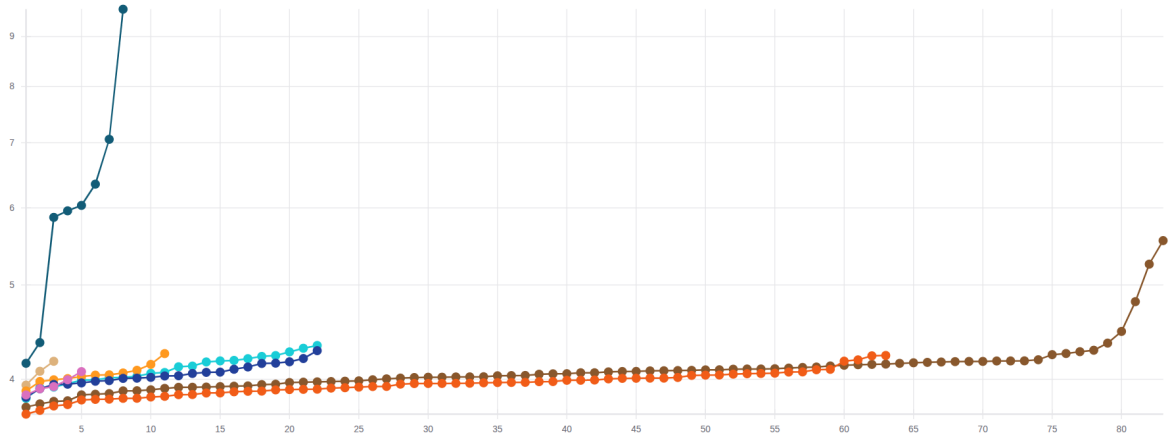


Figure 6.1: Benchmark run per category

CPAchecker arrays
CPAchecker cast
CPAchecker control_flow_structures
CPAchecker function_calls
CPAchecker general_functionality
CPAchecker pointers
CPAchecker struct_and_member_access
CPAchecker ifc-bench

CHAPTER 7

Conclusion and Future Work

7.1 Future Work

In following I present my suggestions for future work as continuation of this thesis.

An extension of the benchmark set could be done as part of future work. The extension would need to contain benchmark cases for scenarios that are not covered in this work, and complement the ones that are only partially covered, such as the ones for type casting. Additionally, the same set of benchmarks could be duplicated, but with assumptions tailored for different type of analysis, e.g., a ground truth for analyze a path-insensitive analysis.

Regarding the implementation, there is still of room for improvement. The main issue of the current implementation is the strong coupling that exist between value tracking and taint tracking. By resolving this, the TaintAnalysisCPA could focus solely in taint propagation, improving the state generation, and avoiding the early termination of the program. This issue causes the bug that we saw in the benchmark program `trippleWhileLoopSafe_1.c` where the analysis considers all the states reached too early, leading to a wrong verdict.

Another point to consider in the future, is to add support for configurable sinks. This would make the analysis more usable in real C programs without the need to "translate" the program to verify. The support of more data types for the SOURCES list is also an aspect that can be improved.

Regarding do-while loops, the missing implementation to support implicit taint propagation is not hard to achieve. The only thing that needs to be implemented for these loops is the recognition of tainted variables controlling the loop, and the recognition of a null value controlling the loop condition. The current solution for while and for-loops does not apply to do-while loops, because the check is based on the incoming and outgoing edges of the relevant node. This recognition of the loop condition and iteration indexes are not in the same entering/leaving edges as in a

do-while loop in different edges. A separate check for do-while loops can be done to ensure the termination and correct taint propagation of do-while loops.
The support for taint propagation related to cast operations is not implemented in this analysis. CPAchecker counts with all the necessary parsing elements to address this without much trouble.
To implement support for implicit taint propagation for `FunctionCallEdge` and `FunctionReturnEdge` would be an improvement which could reduce the probability of falling in infinite execution paths. The complexity of this would be in the inter procedural calls when they are made from inside a control structure controlled by tainted variables, or by non-deterministic conditions.
Finally, supporting recursion would be a very nice improvement, and, since CPAchecker already counts with certain features that support this, it would not be an unimaginable new feature.

7.2 Conclusion

This work contributes with an implementation of taint analysis, capable of effectively tracking the information flow through many structures of C programs in a path-sensitive way considering explicit and implicit information flow, recognizing the influence of variables in many scenarios and reporting possible leaks of information on these programs. Although improvements in several ambits of the implementation can be made, the benchmarking process shows that CPAchecker has the necessary elements to be the backend of a strong taint analysis. Complementary to the development of the taint analysis modul, this work contributes with a benchmark set for testing taint analysis tools. The benchmark set has a well documented ground truth described in this work 5, and additionally in several benchmarks directly in the programs in form of comments. The benchmark set is designed to effectively discover bugs in the core functionality of a path-sensitive taint analysis tool that supports both explicit and implicit information flow. In case someone wants to tests its tool with the benchmark set presented here, it must be clear that there are still uncovered aspects, and that the ground truth in form of assertions is designed for test a tool that complains with the characteristics that we mentioned above. However, the ground-truth can be modified to test, for example a path-insensitive analysis. For that, the assertions, for example, in benchmarks with control structures would have to be made.

Bibliography

[1] D. Beyer, M. Dangl, and P. Wendler. A Unifying View on SMT-Based Software Verification. *J. Autom. Reasoning*, 60(3):299–335, 2018. Published Online: 2017-12-04, Published Print: 2018-03, Update Policy: `https://doi.org/10.1007/springer_crossmark_policy`.

[2] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, July 1977.

[3] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities. Emails: adoupe, bboe, chris, vigna@cs.ucsb.edu.

[4] B. Gruner, C.-A. Brust, and A. Zeller. Finding Information Leaks with Information Flow Fuzzing. *ACM Trans. Softw. Eng. Methodol.*, 34(6):Article No. 184, 18 pages, July 2025.

[5] P. Laud. Semantics and program analysis of computationally secure information flow. In D. Sands, editor, *Programming Languages and Systems*, pages 77–91, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[6] M. Schwarzl, P. Borrello, G. Saileshwar, H. Müller, M. Schwarz, and D. Gruss. Practical Timing Side Channel Attacks on Memory Compression. Cite as: arXiv:2111.08404 [cs.CR] (or arXiv:2111.08404v1 [cs.CR] for this version), DOI: `https://doi.org/10.48550/arXiv.2111.08404`, Affiliations: Graz University of Technology, Sapienza University of Rome, Georgia Tech, CISPA Helmholtz Center for Information Security.

[7] W. Yang, Y. Vizel, P. Subramanyan, A. Gupta, and S. Malik. Lazy Self-composition for Security Verification. pages Published Online: 2018–07–18, Published Print: 2018, Update Policy: `https://doi.org/10.1007/springer_crossmark_policy`. Springer, 2018. Crossref DOI link: `https://doi.org/10.1007/978-3-319-96142-2_11`.