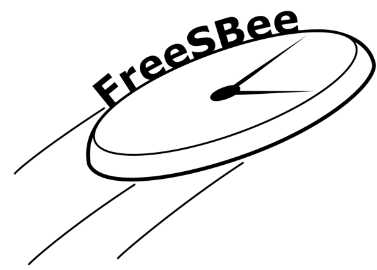

— D2.1 Concept for source-code transformation

Version:	1.01
Publication date:	10.04.2024
Authors:	David Oswald (KAOS)
	Sebastian Reiter (FZI)
	Julian Ganz (FZI)
	David Knothe (FZI)
	Anton Paule (FZI)
	Simon Wegener (AbsInt)
	Stephan Wilhelm (AbsInt)
Contact:	Sebastian Reiter <reiter@fzi.de>

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

– Table of content

1 Introduction 3

1.1 Taint Analysis 3

1.2 Current Workflow 3

2 Taint analysis with Astrée 5

2.1 Taint analysis example 6

2.2 Astrée usage 6

2.3 Reporting of tainted program locations 6

3 Automatic taint annotation 7

3.1 Annotation language 7

3.1.1 Attribute maybe_tainted 7

3.1.2 Attribute propagate_taint 8

3.2 Extraction of taint information 8

3.2.1 Optional source code preparation 8

3.2.2 Extracting taint information from logs 9

3.2.3 Extracting taint information from taint report 9

3.2.4 Introducing annotations into existing source code 9

4 Use of taint annotations in the toolchain 11

4.1 Overview of CompCert 11

4.2 Adding Taint Annotations to CompCert 12

5 References 14

1 Introduction

The aim of the FreeSBee project is the automatic detection and mitigation of timing side-channel attacks in embedded software. A timing side-channel is an attack surface that can arise when the value of secret data like a cryptographic key or a user password has a measurable influence on the execution time of a program. When an attacker knows the workings of the cryptographic function, they may be able to draw conclusions about the secret data by precisely analyzing these variations in execution time over multiple runs. This is not just a theoretical attack but has been demonstrated and exploited many times in the past, as described in our previous report [1].

There are two main ways how a program's execution time can depend on the value of data:

- through data dependence, when operations take different amounts of time depending on the data they operate on (for example through cache effects), and
- through control dependence, when the program performs a different sequence of operations depending on the data (due to secret-dependent conditional statements).

To mitigate the attack surface of timing side-channels, both of these dependencies must be eliminated as far as possible. Therefore, an understanding is required of which data is control- or data-dependent on secret data. The program is then compiled in such a way that all timing variations introduced by any of this secret-dependent data are removed, making the program timing-secure.

In the following, this report describes how *taint analysis* is used to detect secret-dependent data in embedded software and how it is incorporated into the FreeSBee workflow.

1.1 Taint Analysis

Taint analysis, or information-flow tracking, is a well-known code analysis technique that is mainly used to detect software security vulnerabilities like code injection. Taint analysis tracks the flow of data from given *sources*, which is either user-input data or sensitive information like encryption keys, and alerts when such sensitive data enters a *sink* like an SQL statement or an output to the user.

Taint analysis comes in two main forms: dynamic and static. While dynamic taint analysis tags sensitive data in memory during the execution of the program and observes how the data is accessed, static taint analysis considers the program's control-flow graphs and tries to estimate a set of values that could possibly depend on the tainted sources. While dynamic taint analysis only detects values that actually depend on tainted data when running the program – it is under-approximating, meaning that it may miss edge-case values – static taint analysis is over-approximating, meaning it will definitely find all secret-dependent values but will probably mark a few values incorrectly as tainted.

1.2 Current Workflow

Our use of taint analysis in FreeSBee is different to the common use of detecting data flow from sources to sinks: we want to detect secret-dependent conditional statements (that is, if-branches and loops) in the middle-end and secret-dependent time-variant operations in the back-end. We then transform the code to remove those secret-dependent timing variations as much as possible, thereby hardening the program against timing side-channel attacks.

For this, we use the taint analysis capabilities of Astrée [2], a static code analyzer for C and C++ code. Figure 1 shows our envisioned workflow: First, C code[1] is analyzed with Astrée. The generated report is then processed by the taint-annotator tool to annotate the C code with special taint attributes. In the next step, a derivative of the CompCert C compiler digests these taint annotations and uses them to analyze and transform a platform-independent intermediate representation (RTL) into constant-time code. Further platform-dependent analyses and transformations will then be applied in the back-end.

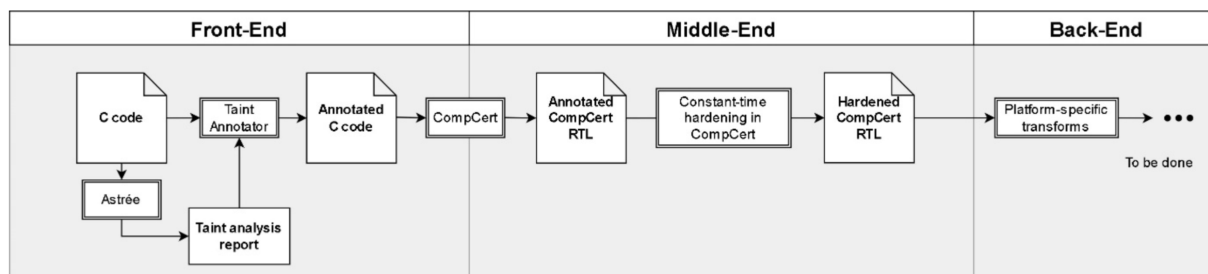

Figure 1: Current and planned toolchain for front-end and middle-end.

[1] We decided to focus on C code in our workflow, noting that this should not be a drawback because a large portion of embedded software is written in C.

2 Taint analysis with Astrée

Astrée's [2] main purpose is to report program defects caused by unspecified and undefined behaviors in C/C++ programs. The reported code defects include integer/floating-point division by zero, out-of-bounds array indexing, erroneous pointer manipulation and dereferencing (e.g., buffer overflows, null pointer dereferencing, dangling pointers), accesses to uninitialized variables, and further sequential programming defects [3]. In addition, Astrée's sound thread interleaving semantics enables it to also report concurrency defects, such as data races, lock/unlock problems, and deadlocks [4].

Astrée soundly over-approximates value ranges and tracks all accesses to variables. It also automatically resolves all data and function pointers. The soundness of the analysis ensures that all potential targets of data and function pointers are taken into account.

Astrée's taint analysis [5] is performed as part of the same whole-program analysis that detects program defects and undefined behaviors. To this end, the concrete semantics of programs are enhanced by program execution with tainting, the formal equivalent of flipping an otherwise unused bit in the dynamic approach to taint analysis. Astrée abstracts this extra information in an efficient and sound way, using a dedicated abstract domain.

Taint analysis then consists in discovering data dependencies using a non-standard semantics of programs, where an imaginary taint is associated to some input values. Considering a standard semantics that uses a successor relation between program states, and considering that a program state is a map from memory locations (variables, program counter, etc.) to values in V, the tainted semantics relates tainted states, which are maps from the same memory locations to $V \times \{taint, notaint\}$, such that if we project on V we get the same relation as with the standard semantics.

Astrée allows the introduction of different hues of tainting. The global semantics is sound for each taint hue in isolation.

Taint analysis in Astrée is enabled by setting the number of tracked taint hues to a value > *0* (via the option *track-taint-hues*) and inserting the following directives into the analyzed code:

- *__ASTREE_taint((variables; taint-hues));* assigns taint hues to variables that act as sources for these taints.
- *__ASTREE_taint_sink((variables; taint-hues));* declares variables as a sinks for the specified taint hues.

Astrée then reports an alarm when a taint spawned by one of the sources reaches a sink for the corresponding hue. Taint sinks also remove the specified taint hues, i.e., they do not propagate through sinks.

Finally, the extra option *taint-control-flow-context* controls whether Astrée's taint analysis also takes control dependencies into account. When enabled, the program locations depending on a tainted condition are considered to be tainted by the condition taints.

Please note that at the point of writing this document, the taint analysis of Astrée is undergoing a major update. Future versions will use new directives for taint sources and sinks and provide additional capabilities for reporting of tainted program locations and for clearing taints.

2.1 Taint analysis example

Here is a small, abstract example of a taint analysis with Astrée. Consider the following program fragment:

```
__ASTREE_taint_add((input; 1));
__ASTREE_taint_sink((output; 1));
auto a = input;          // (1)
f(&b, a);                // f: *b <- a (2)
if (b) ...               // (3)
output = b + ...;        // (4)
```

The input variable is tainted by hue 1. The output variable is declared as sink for taint hue 1. The taint analysis now tracks taint propagation in simple direct cases like (1), indirect flow via pointers as in (2), as well as through complex computation exemplified by (4). Whenever the output taint sink is reached by a taint, an alarm is reported of the kind:

```
   ALARM (D) taint_sink: tainting 4 byte(s) at offset 0 in variable output
with hue 1
```

2.2 Astrée usage

Astrée can be used interactively with a GUI or in batch mode using a command line interface. The latter takes an XML specification (DAX file) that expresses all analysis inputs, settings, and outputs (report files). The main report files can also be specified on the command line.

The taint directives can be included as externally stored analysis configuration, so no actual code modifications are required.

The main output of the analysis are the text and XML report files. These files contain the complete analysis configuration and the analysis results. Astrée's main results are the findings produced by the analyzer, which include alarms about program defects and undefined behavior. If taint analysis is enabled, any cases of taints reaching specified taint sinks also occur in this list of findings.

Additionally, Astrée allows users to specify custom report files. Supported formats are text, csv, and html. The contents of these report files can be configured by users but the tool also comes with a set of pre-defined configurations.

2.3 Reporting of tainted program locations

Although Astrée routinely only reports the program locations at which taints are assigned to sinks, the analysis is aware of all program locations that are reached by taints. The locations can be displayed in tabular form in the GUI and highlighted in the GUI's source code editor.

For the FreeSBee project Astrée was extended such that users can request tainted program locations to be added to its custom report files, e.g. in csv format, for automatic post processing.

3 Automatic taint annotation

For the targeted hardening of software against timing side-channels in the compiler, it needs at least some results of the taint analysis, which can be extracted from reports generated by the Astrée static analysis tool. Making the compiler digest these reports directly, however, would introduce unnecessary and unwelcome coupling in the overall tooling, as the reports are not suited for consumption by a compiler and would require a disproportionate amount of additional logic. Instead, we decided to introduce an additional tool, which processes the reports and distills out the information for the compiler in an easy-to-use format.

As any additional file may require some new, possibly obscure command line options to the compiler which the build system may need to handle, we decided on an interface based on C/C++ attributes. These are embedded into the source code directly and allow re-use after some code-changes while being easy to understand by developers without additional tooling or special training.

3.1 Annotation language

Besides informing the compiler of variables whose values must not affect observable timing, the annotations also pose as an interface through which a developer may take action during the overall process. Primarily, we want to give developers the ability to check whether the annotations introduced are sensible and sufficient, but also to modify, add or remove annotations by hand if warranted by their domain knowledge. This calls for expressive and readable annotations that do not needlessly clutter the source code. We thus chose to keep the number of positions in source code where we may place an annotation low, at the expense of some analysis being necessary during compilation.

This additional analysis, however, can be more lenient in some aspects compared to the comprehensive analysis Astrée needs to perform. For example, we may allow some degree of over-approximation since its effects—fewer opportunities for optimizations—are still preferable to the current state of the art—no or only minimal optimizations allowed for the entire portion of code.

Interprocedural analysis is both expensive and usually not carried out by compilers. Usually, each function in source code is translated to one, static, portion of the resulting binary code during compilation, regardless how often or from how many contexts it is called. Concerning its translation, we thus do not distinguish between calling contexts in which a given function receives tainted values from contexts in which it does not. We therefore consider values either as “potentially tainted” or as not tainted.

In order to achieve our goal of minimizing the number of necessary annotations, we annotate function declarations, in contrast to implementations such as assignments and control flow inside a function body. Since we are interested in tainted values, the annotations need to signal which values entering a function may be tainted. We thus preliminarily defined an attribute “maybe_tainted” which is simply added to function parameters and global variables potentially carrying taint.

3.1.1 Attribute maybe_tainted

During the compilation of a function, the compiler will need to perform a data-flow analysis in order to identify statements and control-flow inside the function that need to be hardened. After considering a variety of code examples, we concluded that, for our purposes, the preliminary “maybe_tainted” attribute was not sufficient: especially in cryptographic code, a parameter may often be a pointer, e.g. either to an encryption key or to private data. We therefore redefined the attribute such that it supports distinguishing between pointer and pointee, as well as selecting certain subsets.

The redefined attribute “maybe_tainted” can be applied to function declarations and takes a list of “taint expressions” specifying potentially tainted values based on parameter names. These expressions borrow from familiar C syntax and allow dereferencing pointers as well as addressing individual struct and array members as usual in C. For concise specifications, we also introduced range expressions in array subscripts with syntax and semantics similar to range indexing in the Rust programming language.

For global variables, we decided to keep the “simple”, parameter-less form of the attribute for the time being.

3.1.2 Attribute propagate_taint

As explained above, a compiler will not generally perform interprocedural analysis. Thus, we need to handle any function call as a black box. Strictly, we would have to assume that any function call could introduce taint, leading to an unacceptable degree of over-approximation. Thus, we decided to make at least some data-flow information, or to be more precise taint-flow information, available via an additional attribute and establish a few heuristics for cases when such an attribute is absent.

In the absence of an attribute, we assume that a function forwards any taint it receives via parameters to its return value and all other “out” parameters, i.e. the pointees of non-const pointers it receives as parameters. However, we assume that a function that is not annotated is free of side effects. That is, that its interaction with the outside is restricted to its parameters and return value. Thus, we assume that such a function does neither “spray” taint it receives over global variables or more generally the program’s memory nor picks up taint from memory locations such as global variables, unless they are explicitly provided via a parameter.

These assumptions are too far-reaching for any analysis generally considered sound. For example, a tool like Astrée is generally not allowed to make such assumptions. However, we consider them reasonable for our use-case, especially considering that any function for which the assumptions do not hold can easily be annotated with an appropriate “propagate_const” attribute.

3.2 Extraction of taint information

Astrée exports usable taint information through two artifacts: logs and a dedicated report, the latter not being available originally but introduced in Astrée version 23.10i after deliberations between project partners during a FreeSBee project meeting. AbsInt introduced this new feature as part of the FreeSBee project.

The logs generated by Astrée include detailed taint information including the full context for specific occurrences of tainted values. However, these log files are not intended for consumption by tools and identification of relevant information amid the sometimes only loosely structured information may not be robust. In addition, logs originally only contained taint information at positions with Astrée directives in the analyzed code. Thus, extracting taint information from the dedicated taint reports is the preferred way.

3.2.1 Optional source code preparation

Because, originally, Astrée directives were necessary for retrieval of taint information, our automated annotation tool includes a feature for introducing a log directive into each function body in existing source code. For this purpose, the tool reads and parses source files and locates all function definitions, i.e. function declarations with a function body. Into each function body it then adds an Astrée directive logging the state of all parameters of the function at very top.

This would ensure that, on each evaluation of a function, Astrée would log the state of each individual parameter before any use in the function body. This would include taint information for the parameter. However, we did not cover global variables in the log statement, as those were not as trivially to detect.

Ultimately, this preparation step only served as a stepping-stone to gather some experience with the tools involved and establish a preliminary workflow for experimentation.

3.2.2 Extracting taint information from logs

Naturally, Astrée logs are simple text files listing log entries, which are dominated by log messages. As the entries are only loosely separated from another with newlines, which is not sufficient in the presence of multi-line-entries, we consider them unstructured data. Astrée also keeps a more structured representation of the logs, which at least clearly separates individual log entries. However, because the format is not stable across releases and the entries in question still need to be filtered and dissected, we decided to extract taint information from the textual representation instead.

When interpreting a log directive, Astrée issues a multiline log entry containing the context, the directive’s source code position and various facts about logged variables such as known facts in domains supported by the static analyzer. Taint analysis is one such domain. Fortunately, the log entry is clearly identifiable as a block encapsulated in brackets (“[” and “]”), with supporting indentation. Thus, we could extract these blocks from log files in a streaming fashion.

For each block, we then try to determine whether it is relevant based on the type of block. For relevant blocks, we then try to extract the context. We specifically extract the immediate call, i.e. the name of the function interpreted when the log directive was reached. We then try to find sections regarding the taint domain, which—like most domain specific sections—are encapsulated in angle brackets (“<” and “>”). This section contains a list of values with their tainted state, which we then associate with the function name (which is unique per source file in C) and the position taken from the block.

3.2.3 Extracting taint information from taint report

The dedicated report has the form of a CSV file with enumerated and named sections separated by empty lines. Some of these sections convey metadata, including analysis configuration. The last section contains taint information: its CSV data declares entities in the form of source code locations and their taint. After identifying this section, we proceed transferring the data into an internal representation using off-the-shelf libraries.

3.2.4 Introducing annotations into existing source code

Given the processed taint information from Astrée, the source code can be annotated based on syntax alone. As a first step of this code-to-code transformation, the automated tool thus reads sources and generates a parse tree. In contrast to an abstract syntax tree (AST), a parse tree represents the source code’s syntactical structure and is generally devoid of the semantics associated with it, and allows for a more precise location of syntactical elements. The latter is, of course, of interest for the transformation.

In order to introduce “maybe_tainted” attributes, we locate all function definitions in the parse tree. For each of those definitions, we look up all taint information relevant for that function name and file. We then correlate that information with the function’s arguments. Depending on the taint information’s concrete nature, this may require some inspection of the function body. The correlated information allows us to generate a “maybe_tainted” attribute, which, together with the target position, is added to a list of insertions for the specific file.

Unlike “maybe_tainted” attributes, “propagate_taint” attributes are relevant during the processing of other functions. Thus, it needs to be attached not to the function’s definition, but its (public) declaration. Mechanically deriving this attribute takes considerably more effort than for the “maybe_tainted” attribute. Given a prior analysis carried out by Astrée, we can partially derive taint flow for *observed* inputs, i.e.

for contexts from which the function was called. In the simplest case, for each context through which a given function received taint, we extract the outputs and generate “propagate_taint” attributed accordingly.

However, the resulting list of attributes will only be valid for observed cases, i.e., we will most likely not achieve the coverage required. In addition, in practice the annotations will need to be generalized. Thus, intervention by a developer will be necessary for the “propagate_taint” attribute.

4 Use of taint annotations in the toolchain

Figure 1 depicts the envisioned toolchain for the front-end and middle-end. The details of the taint-annotations applied in the front-end were described in Chapter 3; this chapter will focus on the connection of the front-end to the middle-end via CompCert.

We use the CompCert compiler to transform C code into a platform-independent intermediate representation (IR): taint-annotated CompCert RTL. Our extension of CompCert will then analyze and harden this IR. The following will give a short overview of CompCert and of how we integrate the taint annotations from Chapter 3 into CompCert.

4.1 Overview of CompCert

CompCert is a C compiler originally developed by Xavier Leroy [6] and maintained and distributed by the project partner AbsInt [7]. CompCert consists of a handful of intermediate representations, each being a little more concrete and hardware-oriented than the previous one, starting from the C abstract syntax tree (AST) and ending with an architecture-specific assembler-like representation. During compilation, CompCert translates the source program into all the different representations step by step and performs various optimizations alongside.

Most of these transformations and optimizations are formally proven to retain the semantics of the code, making CompCert a *verified* compiler: its compilation passes cannot introduce bugs, as they are proven correct. This is achieved by having written most of CompCert in Coq, a proof assistant with an associated programming language that allows formulating and proving facts about the functions that the programmer has written. Still, non-proven parts of CompCert are developed in OCaml, which is compatible with Coq and which is the developing language of our choice.

RTL (register transfer language) is an intermediate representation somewhere in the middle of the CompCert IR chain (see Figure 2) that works with abstract registers (also called temporaries) that can hold arbitrary 32-/64-bit values. An RTL statement is either a k-ary register operation (like addition or move), a memory load/store instruction, a function call or a conditional statement. RTL represents each top-level function as a control-flow graph: nodes are RTL statements and edges describe how control may flow, either unguarded or guarded by a conditional statement. This representation, together with the usage of abstract registers, makes it perfect for complex analysis and transforms like constant-time hardening.

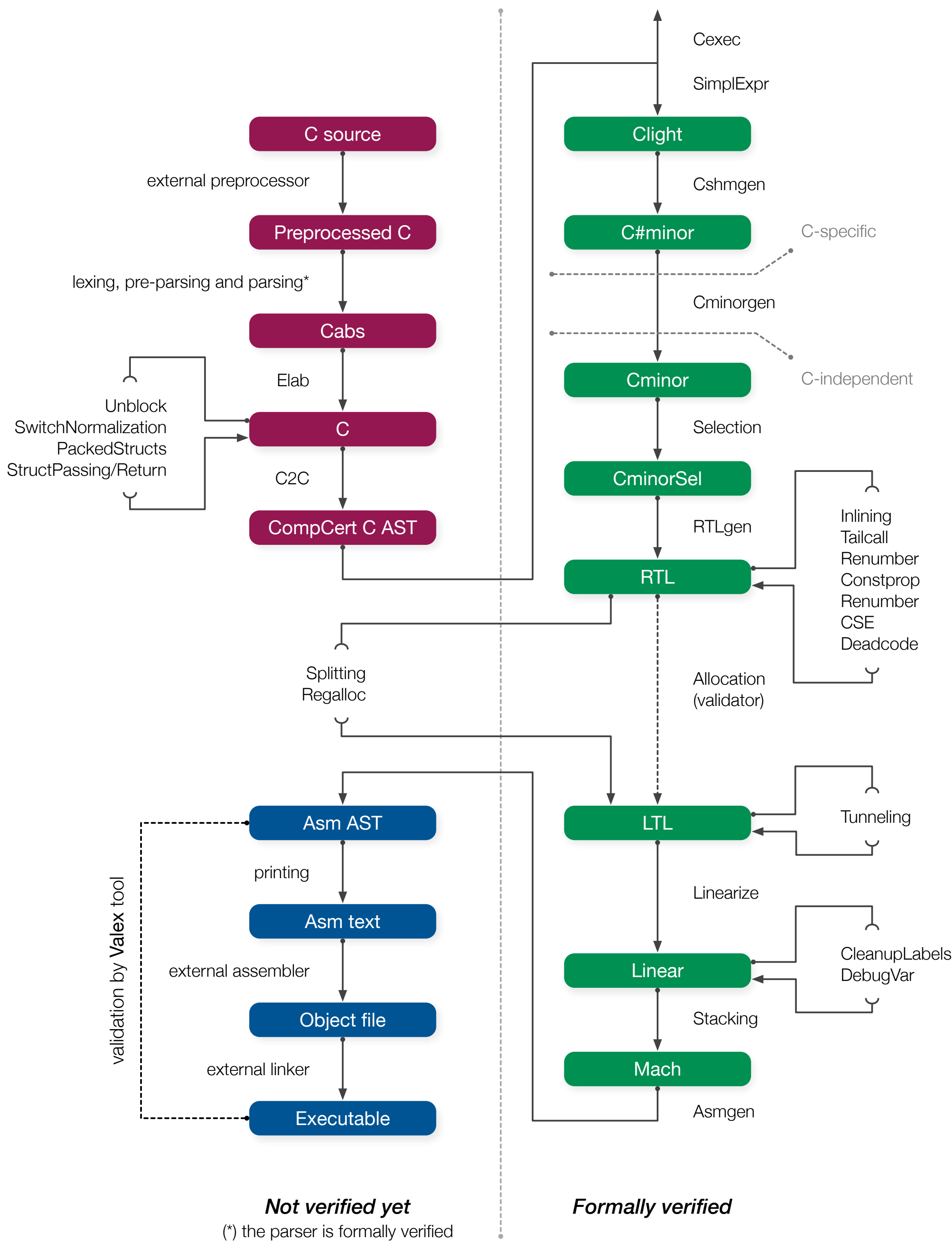

Figure 2: Internal structure of CompCert

4.2 Adding Taint Annotations to CompCert

CompCert, being a C compiler, parses C attributes and attaches them to declarations. This makes it simple to incorporate the attributes from Chapter 3.1 into CompCert.

During *elaboration*, CompCert converts a list of parsed top-level definitions into an abstract syntax tree (AST). This is also the place where specifiers and attributes are checked for correctness and converted

into a custom form. We can attach there to extract and convert the *maybe_tainted* and *propagate_taint* attributes into a value of the *C.attribute* type that we have extended. We then attach it to the IR construct representing the function or the global variable.

To pass the taint information from the C AST along to RTL, we note that most of the intermediate representations of CompCert are built in a very similar fashion: they contain a list of global definitions, which are each either a global variable or a function. While function bodies are always represented a bit differently (as a concrete or abstract syntax tree, as an expression, as a control-flow-graph), many elements like global variables or function parameters are represented similarly. This allows us to pass the taint attributes along the IR chain with very little to no modifications.

For example, the intermediate languages *CSyntax*, *Clight*, *Csharpminor*, *Cminor*, *Cminorsel* all have a *function* record that describes the structure of a function, each of which has an *fn_params* field. These parameters are always represented as a list of identifiers, so we can add a field *fn_taint_attributes* that expresses taint information on the parameter names. Only when converting from *Cminorsel* to *RTL* we have to convert the taint information from being parametrized over identifiers to registers.

5 References

[1] FreeSBee Consortium, „D1.1 State of the Art: Timing and data flow side-channels,“ 2023.

[2] AbsInt Angewandte Informatik GmbH, „Fast and sound static analysis,“ 2024. [Online]. Available: www.absint.com/astree.

[3] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné und X. Rival, „Proving the Absence of Runtime Errors,“ in *Embedded Real Time Software and Systems Congress ERTS²*, Toulouse, 2010.

[4] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm und C. Ferdinand, „Proving the Absence of Run-Time Errors and Data Races with Astrée,“ in *ERTS 2016: 8th European Congress on Embedded Real Time Software and Systems*, Toulouse, 2016.

[5] D. Kästner, L. Mauborgne, S. Wilhelm, C. Mallon und C. Ferdinand, „Static Data and Control Coupling Analysis,“ in *ERTS 2022: 11th European Congress on Embeded Real Time Software and Systems*, Toulouse, 2022.

[6] X. Leroy, „Formal verification of a realistic compiler,“ *Communications of the ACM,* Bd. 52, Nr. 7, pp. 107-115, 2009.

[7] AbsInt Angewandte Informatik GmbH, „Formally verified compilation,“ 2024. [Online]. Available: www.absint.com/compcert.