

Universität Hamburg  
Department Informatik



Diplomarbeit

# Using Compiler-Intermediate Representations for Security-Related Static Analysis

**Thilo Mende**

---

Studiengang Informatik  
E-Mail: [da@thilomende.de](mailto:da@thilomende.de)

Erstgutachter: Prof. Dr. Joachim Posegga  
Universität Hamburg, Department Informatik  
Zweitgutachter: Prof. Dr. Dieter Gollmann  
Technische Universität Hamburg-Harburg

Hamburg, der 07. August 2006



# Abstract

Security-critical defects are pervasive in nowadays software and responsible for a large amount of security incidents. Although modern programming languages eliminate some commonly exploited security problems, new ways to breach software security are published regularly.

Static analysis, the analysis of programs at compile-time, was originally used for compiler optimizations. It has recently been applied in the realm of software security to find commonly exploited security problems. The results are promising: A large amount of today's vulnerabilities may be detected using fully automated tools.

Security-related static analysis and optimizing compilers build on the same foundation. Both perform program analysis at compile time and need a representation of programs that is easy to analyze.

In this thesis it has been analyzed whether it is feasible to use compiler intermediate representations, i.e. data structures used to represent the source program inside the compiler, for security-related static analysis. The goal is to avoid duplicate work in order to accelerate the development of new approaches in this area. Additionally, intermediate representations are often language-independent, allowing the development of approaches for more than one programming language.

Data structures required for different security-related analyses have been identified. The intermediate representations of a wide-spread, open source compiler, the GNU Compiler Collection, have been evaluated regarding their usability for security-related static analysis. The TreeSSA framework is a recently introduced language and target independent optimization framework. Its intermediate representation GIMPLE is designed for high-level analyses and provides most data structures used for security-related static analysis. This framework has been extended to export its intermediate representation. The extension provides access to commonly used data structures for new security-related static analysis approaches in a language-independent way.

Two aspects of the exported data structures were evaluated. A classical static analysis algorithm has been implemented to demonstrate the usability of the proposed solution. The robustness of the extension has been evaluated using three real-world software packages.

It has been found that intermediate representations inside the GNU Compiler Collection are suitable for static analyses aiming at the detection of common security problems. The proof of concept analysis has shown that the exported data structures are valuable for security-related static analysis tools.



# Contents

<b>List of Figures</b>	<b>V</b>
<b>1. Introduction</b>	<b>1</b>
<b>2. Static Analysis for Software Security</b>	<b>3</b>
2.1. Definition . . . . .	3
2.2. A case study: Integer Range Analysis by Abstract Interpretation . . . . .	4
2.3. Prevalent Data Structures for Program Analysis . . . . .	7
2.3.1. Parse Tree . . . . .	7
2.3.2. Control Flow Graph . . . . .	8
2.3.3. Call Graph . . . . .	9
2.3.4. Data Flow Information . . . . .	10
2.4. Static Analysis Tools . . . . .	11
2.4.1. Flawfinder . . . . .	11
2.4.2. BOON . . . . .	11
2.4.3. Metal/xgcc . . . . .	12
2.4.4. MOPS . . . . .	12
2.4.5. Summary . . . . .	12
<b>3. The GNU Compiler Collection</b>	<b>15</b>
3.1. Overview . . . . .	15
3.2. GCC Architecture . . . . .	16
3.3. Optimization Framework . . . . .	17
3.3.1. TreeSSA Architecture . . . . .	17
3.3.2. Static Single Assignment Form . . . . .	19
3.3.3. TreeSSA API . . . . .	20
3.3.4. Interprocedural Optimizations . . . . .	21
3.4. Intermediate Representations . . . . .	21
3.4.1. Register Transfer Language . . . . .	21
3.4.2. GENERIC . . . . .	23
3.4.3. GIMPLE . . . . .	23
3.4.4. Exporting Intermediate Representations . . . . .	25

<b>4. Using the TreeSSA framework for security-related Static Analysis</b>	<b>27</b>
4.1. Motivation . . . . .	27
4.2. Choosing an Intermediate Representation . . . . .	28
4.3. Architecture . . . . .	29
4.3.1. On-disk representation . . . . .	30
<b>5. rxd: A robust XML dumper</b>	<b>33</b>
5.1. Integration into TreeSSA . . . . .	33
5.2. Internal Representation of GIMPLE . . . . .	34
5.3. Implementation . . . . .	35
5.3.1. rxd-control . . . . .	35
5.3.2. rxd-output . . . . .	36
5.3.3. rxd-languagespecific . . . . .	37
5.3.4. rxd-cgraph . . . . .	37
5.4. Data Formats . . . . .	37
5.4.1. GIMPLE Statements . . . . .	38
5.4.2. Function Definitions . . . . .	39
5.4.3. Call Graphs . . . . .	42
5.5. Validation . . . . .	43
<b>6. Discussion</b>	<b>45</b>
6.1. Proof of Concept Analysis . . . . .	45
6.1.1. Requirements . . . . .	45
6.1.2. Implementation . . . . .	46
6.1.3. Intermodule Extension . . . . .	46
6.1.4. Results . . . . .	47
6.2. Practical Evaluation . . . . .	48
6.2.1. rxd and real-world software . . . . .	48
6.2.2. Compilation Time . . . . .	48
6.2.3. Space . . . . .	49
6.2.4. Validation . . . . .	50
6.2.5. Integration into the Build-process . . . . .	50
6.2.6. Robustness . . . . .	50
6.3. Limitations . . . . .	51
6.3.1. Implementation Limitations . . . . .	51
6.3.2. Unsupported Tree Codes . . . . .	51
6.3.3. Object-Oriented Languages . . . . .	51
6.3.4. Annotations . . . . .	52
6.3.5. Intermodule Analysis . . . . .	53
6.4. Related Work . . . . .	53

<b>7. Closing Remarks</b>	<b>55</b>
7.1. Future Work . . . . .	55
7.1.1. Object-oriented Languages . . . . .	55
7.1.2. Link-time Optimization . . . . .	56
7.2. Conclusion . . . . .	56
<b>A. Integer Range Analysis Algorithm</b>	<b>59</b>
<b>B. CD-ROM</b>	<b>61</b>
<b>Bibliography</b>	<b>63</b>



# List of Figures

2.1.	Abstract interpretation of assignment and test nodes . . . . .	5
2.2.	Abstract interpretation of regular junction nodes . . . . .	5
2.3.	Abstract interpretation of loop junction nodes . . . . .	6
2.4.	Statement and its tree representations . . . . .	7
2.5.	Control flow graph . . . . .	8
2.6.	Call graph . . . . .	9
2.7.	Definition-Use and Use-Definition chains . . . . .	10
3.1.	Architecture of GCC prior to version 4.0 . . . . .	16
3.2.	Architecture of GCC since version 4.0 . . . . .	17
3.3.	Architecture of TreeSSA . . . . .	18
3.4.	Conversion into SSA form . . . . .	19
3.5.	Simple function in C and corresponding Register Transfer Language . . . . .	22
3.6.	Simple function in C and corresponding GIMPLE and CFG representation . . . . .	24
4.1.	Proposed architecture . . . . .	30
5.1.	Integration into TreeSSA . . . . .	34
5.2.	Rxd's architecture . . . . .	35
5.3.	Format for translation units . . . . .	37
5.4.	Format for functions . . . . .	40
5.5.	Source code used in following examples . . . . .	40
5.6.	Example of rxd's control flow format . . . . .	41
5.7.	Format for call graphs . . . . .	42
5.8.	Example of Rxd's call graph format . . . . .	42
6.1.	Result of an integer range analysis . . . . .	47
6.2.	Method invocation in Java and GIMPLE . . . . .	52

*List of Figures*

# 1. Introduction

Security-critical defects are pervasive in today's software. During the first half of 2006, 3997 vulnerabilities were reported to the CERT coordination center, a security incident response team.<sup>1</sup> Although this number contains duplicates and includes vulnerabilities with different impacts, the vast amount of reports is alarming. The majority of vulnerabilities stems from well-known problems, such as buffer overflows. Modern languages render many of these attacks impossible, but a large amount of legacy systems is still vulnerable. Additionally, new attack methods are constantly published, some of them being conceptual problems that are independent from programming languages.

The approach to publish patches for known vulnerabilities is not satisfying. It is way too expensive for both software vendors and customers. Furthermore, systems are vulnerable at least between the disclosure of a defect and the availability of a patch. Detection of bugs during the development is thus desirable for both parties.

Many vulnerabilities can be avoided by good programming practice, but the amount of possible attacks makes it hard for a single developer to obey all of them. One possible approach are code reviews, but these are cost-intensive and dependent on the skills of the reviewer.

Recently, a set of tools has been developed to detect common programming errors automatically. They perform *static analysis*, i.e. the program is analyzed at compile time. This class of tools, also called *static checkers*, targets the analysis of real-world programs and is supposed to be usable by regular developers. The results of these tools are promising especially because all possible paths inside a program are analyzed.

Developing new static analysis approaches requires tremendous efforts. Many engineering tasks are necessary beside the development and implementation of new algorithms. The program has to be represented in a way that is easy to analyze. Therefore, the static checker has to be able to parse the input-language and to generate additional data structures, such as control flow graphs. This is a major task on its own for real-world programming languages.

The main application of static analysis is code optimization inside compilers. This optimization is performed on *intermediate representations*, i.e. data structures inside the compiler that represent the input program. These are often language-independent, meaning that several programming languages are transformed into the same intermediate representation. This allows the implementation of language-independent optimizations.

The goal of this thesis is to evaluate whether intermediate representations inside a compiler are reusable for security-related static analysis. This could accelerate the implementation of new, possibly language-independent static checkers. Additionally, it

---

<sup>1</sup><http://www.cert.org/stats/>

## 1. Introduction

allows a tight integration of static checkers into the complete development cycle.

In this thesis, the GNU Compiler Collection (GCC) is used as a basis. This is justified as follows: First, it is the only open source compiler that is in large-scale production use. It supports several programming languages and is used for the vast majority of open source software. This facilitates the integration of static checkers into existing build-systems.

This thesis is organized as follows: Security-related static analysis is introduced in Chapter 2. Fundamental problems of static analysis and the prevalent data structures are described. Different approaches of static checkers and their data structure requirements are exemplified using four existing tools.

An introduction to the GNU Compiler Collection can be found in Chapter 3. The general architecture of the compiler and its optimization framework are described together with intermediate representations being used.

The combination of security-related static analysis and compiler intermediate representations is discussed in Chapter 4. At first, the goal of such a combination is further refined. The available intermediate representations are evaluated, and an architecture to accomplish the combination is proposed.

An extension to the GCC has been implemented that provides access to one intermediate representation. The implementation of this extension and the resulting data formats are described in Chapter 5.

An evaluation of the proposed solution is given in Chapter 6. A proof of concept analysis validates the usability of GCC's intermediate representations for static checkers. The robustness of the extension is evaluated using three real-world programs. Additionally, current limitations and related approaches are discussed.

A summary of the results may be found Chapter 7 along with possible extensions.

## 2. Static Analysis for Software Security

The term static analysis refers to a set of program analyses applied for example in the area of compiler optimizations or the detection of security vulnerabilities. A variety of different techniques is used, from simple pattern matching to the application of formal methods. The purpose of this chapter is to introduce basic concepts and limitations of static analysis and explore data structure requirements of different approaches.

This work focuses on security-related static analysis, a detailed definition of the covered range can be found in Section 2.1. A brief introduction of a specific program analysis framework, namely abstract interpretation, along with an example analysis is given in Section 2.2. Although a broad range of approaches and techniques exists, the data structures that are used are often very similar. The prevalent data structures are described in Section 2.3. Some approaches of security related static analysis tools are outlined in Section 2.4, with focus on their data structure requirements.

### 2.1. Definition

The BCS SIGIST <sup>1</sup> defines static analysis as the

Analysis of a program carried out without executing the program.[Bri06]

and thus emphasizes the contrast to dynamic analysis, where the behaviour of program is observed while it is executed.

An example for static analysis is constant propagation inside a compiler. If it is known at compile time that a certain variable has a constant value, the compiler may replace all occurrences of this variable by that constant value. This problem is, as most static analysis problems, undecidable. [Lan92] provides detailed information. In consequence, no analysis can be sound and complete. I.e. the analysis is able to find all errors of one class, and none of these is a false positive. A discussion about the implications and necessary trade-offs is given in [XNHA05] and [CDW04].

Despite this computational drawback, static analysis has very interesting properties: In contrast to dynamic analysis, the results of a static analysis are valid for all possible executions. This is especially valuable in the realm of software security. It is often sufficient for a successful attack to have one possible execution path with an error occurring. In addition, a dynamic analysis must be able to execute code and observe its behaviour, which can be difficult e.g. for operating systems. For a static analysis it is sufficient to be able to compile the code.

---

<sup>1</sup>British Computer Society, Specialist Interest Group in Software Testing

Generally, two different scopes of analysis are distinguished: *Intraprocedural* analysis handles each function independently. *Interprocedural* analysis takes the calling relationships between different functions into account. In most cases, the latter is required, especially for object-oriented languages, where many short and simple procedures are used.

The focus of this work is on approaches that detect security vulnerabilities in real-world software. This class of tools is sometimes referred to as *static checkers* [ASU88]. An introduction to different tools and their approaches is given in Section 2.4.

None of these tools performs a formal verification of the software in the sense that it is proved that the implementation meets its specification. Instead, the tools detect common security problems, e.g. the usage of vulnerable functions or the violation of system specific rules.

A severe problem for all static analyses are false positives. A tool is not effective, if it reports many possible errors, with most of them being false positives. In order to eliminate false positives some of these tools perform data flow analysis to prune infeasible paths or to calculate the possible runtime values of a variable at a certain point. Since most of these problems are, as already noted, in general undecidable a way to handle false positives is necessary. Some approaches are described in [HCXE02].

### 2.2. A case study: Integer Range Analysis by Abstract Interpretation

An algorithm to determine possible run-time values of integer variables at compile-time based on the theory of abstract interpretation is introduced in this section. Although this algorithm is a theoretical example and not directly applicable to real-world software, it exemplifies which problems and requirements arise in program analysis. This algorithm is also used in Section 6.1 to evaluate the solution presented in this thesis.

Abstract interpretation offers a theoretical framework of approximation and is applied successfully in program analysis. A detailed introduction to abstract interpretation, along with its theoretical foundations, can be found in [CC77a] or [NNH99]. One specific analysis, namely integer range analysis as introduced in [CC76] is described here. The goal of this analysis is to determine possible run-time values of integer variables at compile time to be able to omit unnecessary run-time checks for array accesses. Several other security-related applications, e.g. integer overflow detection, are possible.

A copy of the algorithm from [CC76] can also be found in Appendix A. It approximates the possible run-time values of each integer variable (concrete value) using an interval of integers (abstract value). An example: the possible run-time values  $\{0, 1, 5\}$  are approximated by the interval  $[0, 5]$ .

Programs are represented using flow graphs. Three different types of nodes, assignment, test junction nodes, are distinguished. Each edge in the flow graph has a *context*, i.e. a set of variables with their associated abstract values. In the beginning, all contexts are initialized to the empty context.

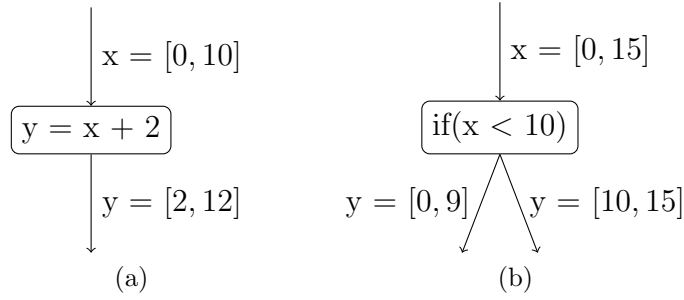


Figure 2.1: Abstract interpretation of assignment (a) and test nodes (b)

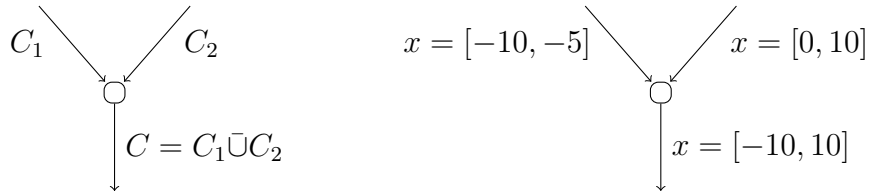


Figure 2.2: Abstract interpretation of regular junction nodes

Starting from the input edge, the algorithm iteratively interprets the effect of each node to its input contexts and thus determines the context of each outgoing edge. Assignment nodes are interpreted using a standard interval arithmetics, an example is given in Figure 2.1(a). Test nodes have two outgoing edges, and their contexts are also determined using interval arithmetics, as depicted in Figure 2.1(b).

Junction nodes merge several possible execution paths introduced by test nodes. The algorithm expects that a graph analysis already identified loops in the flow graph and that a minimal number of junction nodes is marked as loop junction nodes, so each cycle in the flow-graph contains at least one loop junction node.

An operation on intervals equivalent to the union of sets can be defined as

$$[a_1, b_1] \cup [a_2, b_2] = [\min(a_1, a_2), \max(b_1, b_2)] \quad (2.1)$$

This operator is used to merge the different input contexts of a regular junction node as depicted in Figure 2.2.

An algorithm that uses this union operator to interpret loop junction nodes does not necessarily terminate. Instead, a *widening operator*  $\bar{\cup}$  is used as shown in Figure 2.3.

This widening operator has to have two properties that can informally be defined as:

- The result of two intervals combined using the widening operator is bigger or equal to the union of these two intervals,
- the iterative application of the widening operator eventually stabilizes.

The theoretic foundations and a formal definition of these properties are given in [NNH99]. An example is the easiest way to understand this concept for this specific algorithm. An

## 2. Static Analysis for Software Security

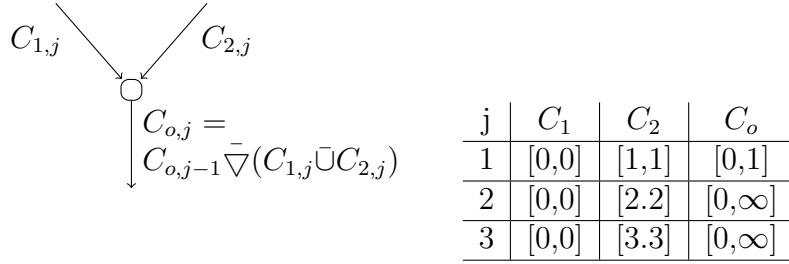


Figure 2.3: Abstract interpretation of loop junction nodes

operator that meets the requirements is

$$[a_1, b_1] \bar{\nabla} [a_2, b_2] = \begin{cases} \text{if } a_2 < a_1 \text{ then } -\infty \text{ else } a_1, \\ \text{if } b_2 > b_1 \text{ then } \infty \text{ else } b_1 \end{cases}$$

The application of this widening operator to evaluate loop junction nodes is demonstrated in Figure 2.3.

An alternative widening operator is proposed in [NNH99]. It is defined as

$$[a_1, b_1] \bar{\nabla} [a_2, b_2] = [\text{LB}(a_1, a_2), \text{UB}(b_1, b_2)]$$

where  $K$  is a finite set of integers and

$$\text{LB}(a_1, a_2) = \begin{cases} a_1 & \text{if } a_1 \leq a_2 \\ k & \text{if } a_2 < a_1 \text{ and } k = \max(k \in K | k \leq a_2) \\ -\infty & \text{if } \forall k \in K : a_2 < k \end{cases}$$

$$\text{UB}(b_1, b_2) = \begin{cases} b_1 & \text{if } b_2 \leq b_1 \\ k & \text{if } b_1 < b_2 \text{ and } k = \min(k \in K | b_2 \leq k) \\ -\infty & \text{if } \forall k \in K : k < b_2 \end{cases}$$

$K$  is for example the set of literal integer values inside a program. This operator is less coarse than the previous one, but an algorithm using it may require more iterations to terminate.

An example analysis of this algorithm using the second widening operator can be found in Section 6.1.

The algorithm described here uses a coarse abstraction, e.g. it is not possible to express that a variable is not zero, or to express relationships between variables. Additionally, programs have to be very simple to be analyzed, e.g. procedures or aggregate types are not modeled. This is not an inherent limitation of abstract interpretation. More accurate abstractions are available, and the Astrée static analyzer [CCF<sup>+</sup>05] demonstrates that this theory is applicable to real-world software.

This algorithm exemplifies some requirements of program analysis in general. Flow graphs are used to represent programs. Loops have to be identified in this graph to avoid non-termination. Statements inside the flow graph have to be evaluated, thus their representation should facilitate this.

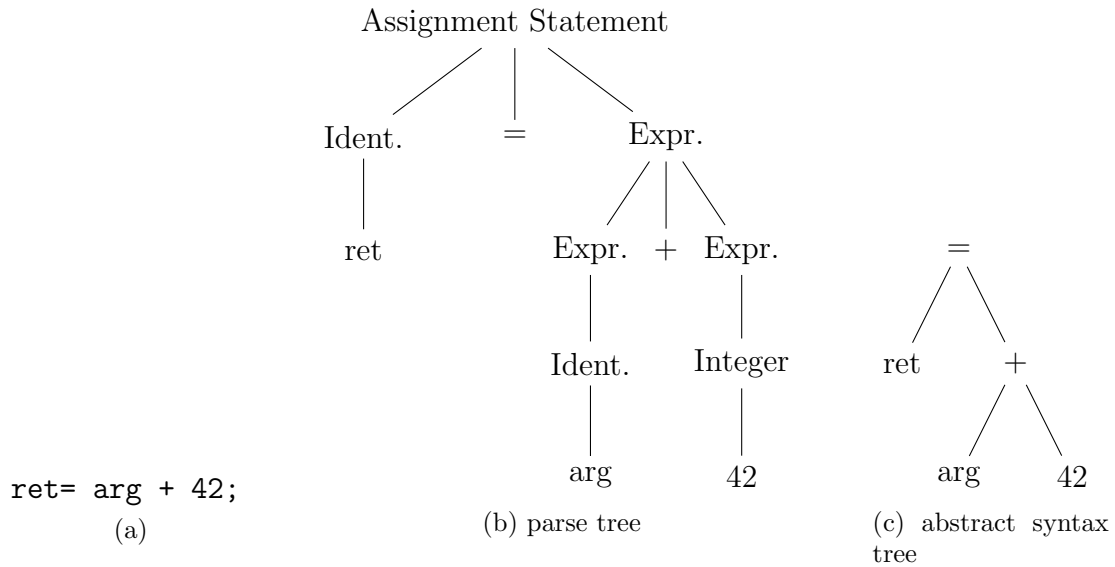


Figure 2.4: Source code (a) and corresponding tree representations (b),(c)

## 2.3. Prevalent Data Structures for Program Analysis

The main data structures for different program analyses, e.g. used inside a compiler or a static checker, are often very similar. The most commonly used are presented in the following section. Details about the generation of these data structures can be found in [ASU88] or [Muc97], this section discusses only their possible usage and fundamental problems that arise during creation. The notation of [Muc97] is used in Section 2.3.2 and Section 2.3.3.

### 2.3.1. Parse Tree

The source code of a program is adequate for analyzing. It is not always easy to determine whether a string in the source code is a variable name, a function call or just part of a comment. Other problems when analyzing source code are typing and scoping of variables.

Therefore, the source program is often translated into a token stream and a syntax analyzer generates a parse tree representing the syntactical structure of the program. Branches in the tree represent production rules of the underlying grammar, leaf nodes represent actual statements.

An abstract syntax tree (AST) is a compact representation of a parse tree where operators are used as interior nodes and the operands are used as their children. For static analysis these two representations are interchangeable. Usually the parse tree is converted to an AST to remove superfluous information. An example of a parse tree and the corresponding AST is given in Figure 2.4.

The tree representation is easier to analyze than the original source code because

```

1  int f(int m){
2    int i = 0;
3    if(m <= 0)
4      return 0;
5    else{
6      while(m > 0){
7        i = i+1;
8        m = m-1;
9      }
10   }
11   return i;
12 }

```

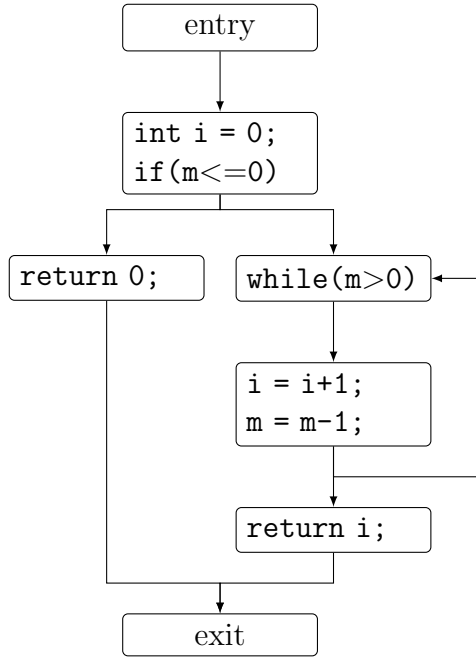


Figure 2.5: Control flow graph

parsing ambiguities are resolved and the meaning of a token is made explicit. If one is interested in all function calls, it is much easier to check whether a node in the syntax tree has the type “call statement” than to use patterns to describe how function calls may be represented in the source code. Another advantage of using a tree representation is that often unique numbers are used to identify variables. The analyzer does not have to handle lexical scopes whenever this unique number is used to identify variables.

### 2.3.2. Control Flow Graph

Static analysis techniques are often not interested in the sequence of statements the in source code but in the possible execution paths inside a function. A commonly used data structure to address this fact is the Control Flow Graph (CFG). It is a graph  $G = \langle N, E \rangle$  where  $N$  is a set of basic blocks. A basic block is a sequence of non-branching statements that can only be entered at beginning and left at the end. The set of edges  $E \subseteq N \times N$  represent the possible flow of control between basic blocks. Alternatively,  $N$  may be the set of single statements inside a function. The statements inside the nodes are often represented as ASTs or other analytical friendly representations. An example of a CFG for a simple function is depicted in Figure 2.5.

The generation of a CFG first identifies basic blocks and connects them according to the control statements inside the function. This gets difficult when more advanced control flow structures such as exception handling or the `setjump` operation are used, one way to approach this problem is described in Section 3.4.3.

Using a CFG, it is easy to get all possible paths through a function or to detect loops

```

1  int f(int i){
2    return g(i) + h();
3  }
4  int g(int i){
5    if(i > 0)
6      return h();
7    else
8      return i();
9  }
10 int h(){
11   return 0;
12 }
13 int i(){
14   return g(1);
15 }
16

```

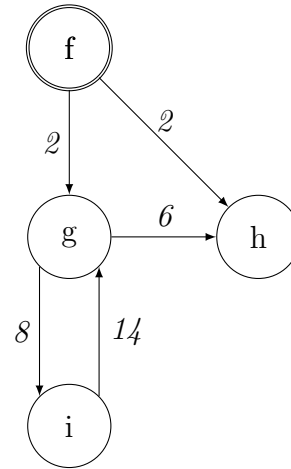


Figure 2.6: Call graph

inside a function, even if these loops are not explicit in the source code, e.g. through if and goto statements.

### 2.3.3. Call Graph

A Call Graph represents the caller-callee relationships inside a program as a directed graph. It can be defined as  $G = \langle N, S, E, r \rangle$  where  $N$  is the set of functions,  $S$  the set of call-site labels and  $E \subseteq N \times N \times S$  the edges representing a function call. Since one function may call the same function more than once, the edges are labeled to identify the relevant call statement for that edge. In [Muc97]  $r \in N$  is defined to be the distinguished entry point to a program. Many applications have more than one possible entry point, especially from the perspective of a security-related analysis, where e.g. every external input may be regarded as an entry point. An example of a call graph can be found in Figure 2.6. Here line numbers are used as call-site labels. However, this is not practical for high-level languages where the same function may be called multiple times in one source code line.

Generating a call graph for languages without procedure-valued variables is easy. Separate compilation makes the construction of a whole-program call graph slightly more complicated, although algorithms to merge call graphs from two compilation units are easy to develop.[Muc97]

A more severe problem is introduced when procedure-valued variables are allowed. This is the case when function pointers are supported, or for method calls in object-oriented languages. Often it is not known until run-time which procedure will actually be called. Therefore the computation of an exact call graph is not always possible

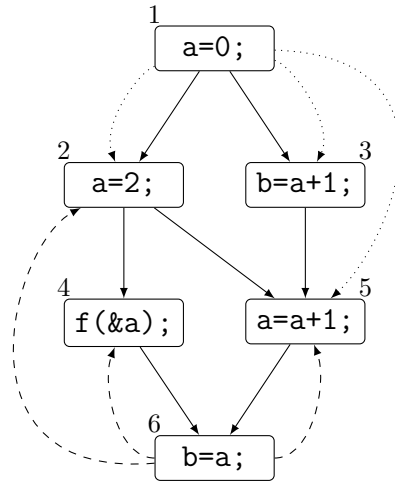


Figure 2.7: Definition-Use and Use-Definition chains:  $du(a, 1) = \{2, 3, 5\}$  and  $ud(a, 6) = \{2, 4, 5\}$

and approximations for the real run-time call graph are generated. Different trade-offs between speed and precision are discussed in [GDCC97].

### 2.3.4. Data Flow Information

A variety of different data flow information exists and there is no common data structure to represent them because the contained information is diverse. A broad range of data flow information analyses can be found in [Muc97], one of them is the *Reaching Definitions* (RD) analysis.

RD analysis determines which definitions of a variable may reach which points inside a program. A definition reaches a point  $p$  if there is an execution path inside that function from the definition to point  $p$ . This information can be used for instance to perform dead code elimination. The information gathered from a RD analysis can be used to generate use-definition(ud) and definition-use (du) chains that represent the same information in a more compact way. A ud-chain for a variable  $a$  in a given point  $p$  ( $UD(a, p)$ ) determines which definitions of  $a$  may reach point  $p$ . A du-chain links the definition of a variable with all possible usages of that definition. Figure 2.7 depicts this concept.

Another data structure is the program dependency graph (PDG) which represents control and data dependencies inside a function. An algorithm to generate a PDG can be found in [FOW87]. It uses a control flow graph to determine control dependencies and reaching definitions analysis to determine data dependencies. The system dependency graph (SDG) described in [HRB88] is the interprocedural equivalent of a PDG.

## 2.4. Static Analysis Tools

In this section tools are introduced that perform security-related static analysis on real-world software. The goal is to understand which of the previously introduced data structures are commonly used, and which additional data structures might be required.

The amount of different static analysis approaches makes an exhaustive list infeasible, thus four tools representing different techniques have been chosen. References to tools working in a similar way are provided.

### 2.4.1. Flawfinder

A lot of security vulnerabilities arise from the usage of standard functions that are difficult to use securely. Plenty of these errors can be eliminated by good programming practice, i.e. validation of input data, or by using secure replacements for these functions.

This observation is exploited by FlawFinder [Whe06]. It compares source code with a database of vulnerable code patterns and informs the developer which parts of the code may be insecure. Additional information how this problem could be avoided is provided. The database contains patterns detecting likely buffer overflows, format string vulnerabilities, usage of weak random number generators and Time-of-check-time-of-use problems. RATS [Sec06] and ITS4 [VBKM00] work in a similar way.

All of these tools operate directly on the source code, which requires quite complex regular expression to detect function calls and their parameters reliably. This would be much easier if a tree representation of the input program was used. [VBKM00] mentions two reasons why ITS4 does not use a tree-representation. The first one is that ASTs are not able to represent conditional compilation through the usage of preprocessor macros. The second drawback that is mentioned is that real parsing is too slow to provide good interactivity in integrated development environments, such as emacs.

### 2.4.2. BOON

Many real-world exploits in C programs today are buffer overflows through insecure handling of strings. BOON [WFBA00] tries to automatically detect these buffer overflows at compile time.

BOON works intraprocedural and traverses the parse tree of a function to generate an integer constraint system. This approach is flow-insensitive, i.e. all control flow statements are ignored. Integer variables are modeled as intervals and string variables as two integers, one for the number of bytes currently allocated and currently used. This constraint system is later solved by an integer range analysis algorithm reporting possible overflows.

The constraint system could also be generated using an abstract interpretation, similar to the one described in Section 2.2 although [WFBA00] claims higher scalability for BOON's approach.

### 2.4.3. Metal/xgcc

Metal is a language to provide programmer written rules that must be obeyed for correct system behaviour. These rules may then be checked by the compiler extension `xgcc` and the programmer is informed whenever one of the rules is violated. This system is described as an intraprocedural version in [ECCH00] and has later been extended to an interprocedural version [HCXE02].

Rules are defined as state machines where states describe the states of variables and transitions are triggered by source code patterns.

The analysis starts at the root(s) of the call graph and traverses all functions in depth-first order. Each function is represented as a CFG. ASTs are used to represent statements inside basic blocks. `xgcc` follows all execution paths inside a program, interprets the effect to the state machines and raises an error if there is a path that leads to an error state in the state machine.

The approach is unsound to limit the amount of false positives, but shows impressive results and effectiveness for several large real-world software projects. For example, the tool was able to find plenty of memory-allocation errors inside the Linux kernel. [ECCH00]

A similar graph based pattern matching is proposed in [Wil05]. Program Dependency Graphs are used to model security properties and later matched against a PDG/SDG representation of the program. A first implementation of this approach is discussed in [WF05].

### 2.4.4. MOPS

MOPS [CW02] was developed to check at compile time whether a program can violate temporal safety properties. Several security properties that can be checked with MOPS are given in [CDW04], one example is that a `setuid-root` program should drop root privileges before executing an untrusted program.

A security property is described as a finite state automaton. MOPS uses model checking, i.e. it exhaustively searches all possible control flow paths, to verify that there is no path in the program under test that ends in an error state of the automaton. MOPS uses control flow graphs to generate push-down automatons that are used inside the model checker to verify the security properties. ASTs are used inside the CFG.

Many other tools use model checking to find bugs in software, an example is GMC2 [GaSJS05]. Although the approaches have different emphases and use different techniques, the data structure requirements are very similar.

### 2.4.5. Summary

The data structure requirements of the previously discussed static checkers are summarized in Table 2.1. The main data structures used in today's static checkers are tree-based representations and control flow graphs. Although MOPS does not use a call graph to perform interprocedural analysis, an explicit representation of the calling relationships

Table 2.1: Summary of data structure requirements

Tool	AST	CFG	CGRAPH
FlawFinder	possible	-	-
BOON	yes	-	-
xgcc	yes	yes	yes
MOPS	yes	yes	possible

might be useful. When a call graph is used, it is much easier to optimize the ordering of analysis and to handle recursive functions.

The abstract interpretation in Section 2.2 requires a CFG, ideally containing loop information, and an easily accessible representation of statements, e.g. ASTs.

Most of the approaches described here do not use data flow information in their basic analysis method. However, data flow information is computed to prune infeasible execution paths and thus avoid false positives. Easy access to standard data flow information, e.g. reaching definitions, might be useful.

Another analysis regarding data structure requirements can be found in [Wil05]. The focus there is on internal representations that are used during the analysis, and not on the data structures required to generate these representations. Therefore the results are not directly comparable.

## 2. *Static Analysis for Software Security*

## 3. The GNU Compiler Collection

The GNU Compiler Collection (GCC) [Fre06a], started in 1987 as the GNU C Compiler, is an open-source compiler suite. The current release, version 4.1, supports C, C++, Objective-C, Fortran, Java and Ada on over 30 different architectures. GCC is the standard compiler used in Linux and the whole BSD-family including Mac OS X. It is also available on most UNIX platforms and MS Windows.

The purpose of this chapter is to provide an overview of the internals of GCC, focussing on intermediate representations. A general introduction into operation modes of GCC is given in Section 3.1. The architecture of the compiler is the topic of Section 3.2. The optimization phase is described in Section 3.3 while intermediate representations used inside the GCC are introduced in Section 3.4.

### 3.1. Overview

GCC is usually invoked using the so called driver `gcc`. The driver's tasks are to control the preprocessing, compiling, assembling, and linking of one or more input files. The compilation itself is performed by a language-dependent compiler executable e.g. `cc1` for C. A new compiler is started for each translation unit, i.e. for each file in C. The detailed architecture of the compiler itself is described in the next section. Its tasks can be summarized as parsing, optimization and code generation. Traditionally, these steps are performed for each function on its own to reduce memory consumption. All optimizations in this mode are thus intraprocedural.

The unit-at-a-time mode, introduced in version 3.4, performs parsing, optimization and code generation once for each translation unit. Although this requires more memory, it enables interprocedural optimizations. Since a new compiler is started for each translation unit, only functions from the current unit are accessible. This limits interprocedural optimizations to functions inside the compilation unit. E.g. it is not possible to inline a function from a different unit.

The term *intermodule analysis* is used to emphasize that an analysis is interprocedural and operates on several compilation units at once. GCC currently supports a limited form of intermodule optimizations for C programs. Several compilation units can be combined and treated as one unit so that parsing, optimizing and code generation is performed on these units together. This approach is very memory-consuming and not supported for other input languages. Hence, the unit-at-a-time mode is assumed in the following discussions. Intermodule analysis as currently planned for GCC, and its impact on this work, are subject of Section 7.1.2.

### 3. The GNU Compiler Collection

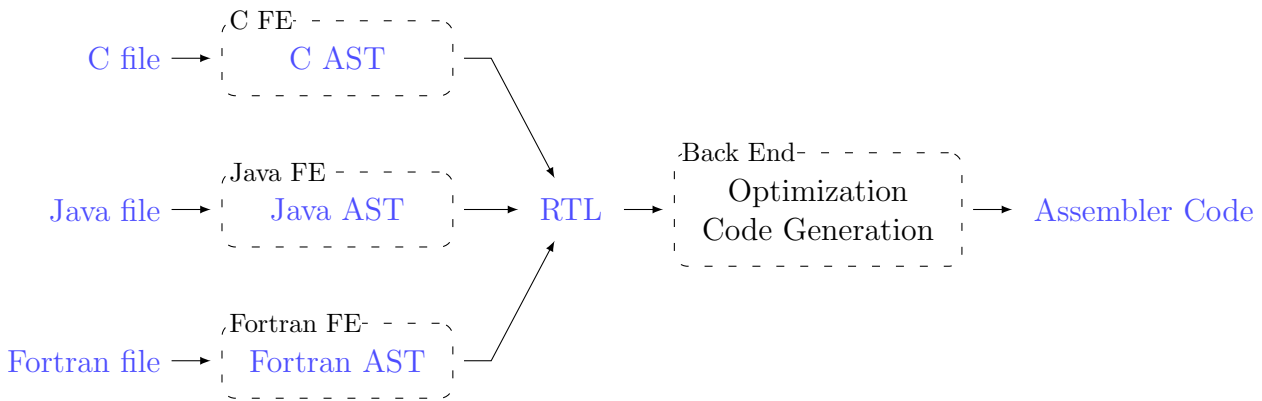


Figure 3.1: Architecture of GCC prior to version 4.0

## 3.2. GCC Architecture

Compilers can be separated into a language specific front end and a target dependant back end.[ASU88] At first, the front end performs lexical and syntax analysis to get a hierarchical representation of the input program. Afterwards checks for semantic and typing errors are carried out. The back end performs then optimizations and generates object or assembler code for the target machine.

Some modern compilers include an additional language and target independent optimization phase sometimes referred to as the *middle end*<sup>1</sup>. High-level optimizations, such as loop optimizations, are performed here. Low-level, target-dependent optimizations, such as instruction scheduling, are performed in the back end. The main difference between a middle end and optimizers inside the back end is the intermediate representation, i.e. the data structures representing the input program. The middle end uses higher-level representations. Therefore array references are supported and type information is available. The back end usually uses a representation close to the target machine, e.g. access to arrays is transformed to offsets to pointers and only few type information are available. More details about different intermediate representations can be found in Section 3.4.

GCC followed the two-phase approach until version 3.4. A flow chart depicting this architecture can be found in Figure 3.2. The front end parses each statement of the source language into a language-dependent parse tree. This tree is translated to the Register Transfer Language (RTL) and transferred to the back end. Optimizations are performed on RTL and assembler code for the target platform is generated. RTL is a low-level intermediate representation similar to an assembly language for a virtual processor. Details are discussed in Section 3.4.1. RTL is suitable for optimizations close to the target machine whereas higher-level optimizations are difficult or infeasible to implement.[Nov03]

Information useful for higher-level analyses are available in parse trees inside each

<sup>1</sup>“Consistency in naming conventions led to this unfortunate term.”[Nov06a]

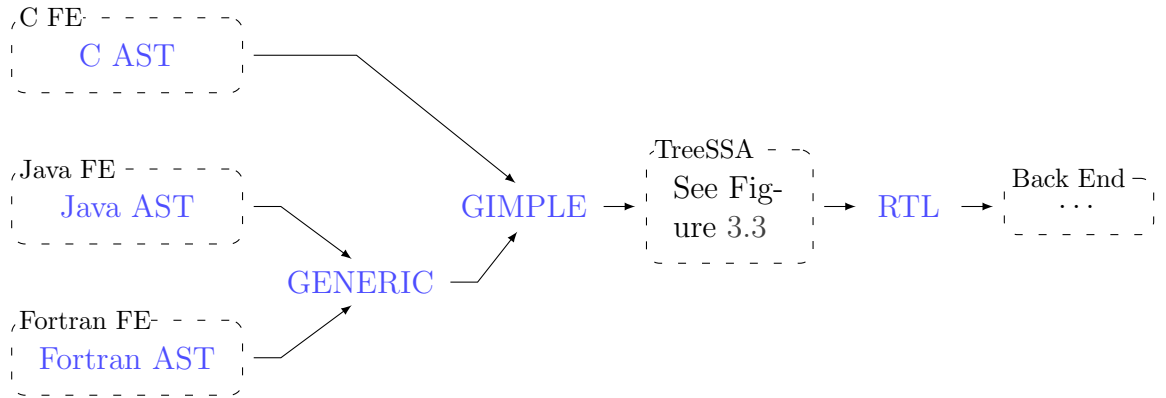


Figure 3.2: Architecture of GCC since version 4.0

front end, although in a language dependent way. Optimizers operating on parse trees thus have to implement an analyzing infrastructure, e.g. to generate CFGs on their own.

### 3.3. Optimization Framework

The TreeSSA framework introduced in version 4.0 provides a high-level, language and target independent optimization infrastructure, a middle end. The goal is to address the limitations arising when RTL is used for optimizations. Low-level ones still uses RTL while high-level optimizations use tree-based representations in static single assignment (SSA) form (see Section 3.3.2).

The integration of the TreeSSA framework into GCC is depicted in Figure 3.2. Two tree-based representations, namely GENERIC and GIMPLE, are introduced. Optimizations are performed on GIMPLE. GENERIC is only used to interface front ends and the optimization phase. A detailed discussion and the reasons for employing two languages can be found in Section 3.4.

#### 3.3.1. TreeSSA Architecture

The architecture of the framework itself can be found in Figure 3.3. The *gimplifier* translates either GENERIC or language-dependent parse trees into GIMPLE. This is transferred to the TreeSSA framework. The interprocedural analysis (IPA) module builds a call graph and performs interprocedural optimizations such as function inlining. The pass manager controls the generation of CFGs and the transformation to SSA form. Intraprocedural optimizations, such as dead code elimination and constant propagation are performed here. Due to data structure restrictions only one function at a time can be in SSA form thus interprocedural optimizers on SSA are currently not available. The compilation unit is finally converted to a non-SSA form and translated to RTL, which is transferred to the back end.

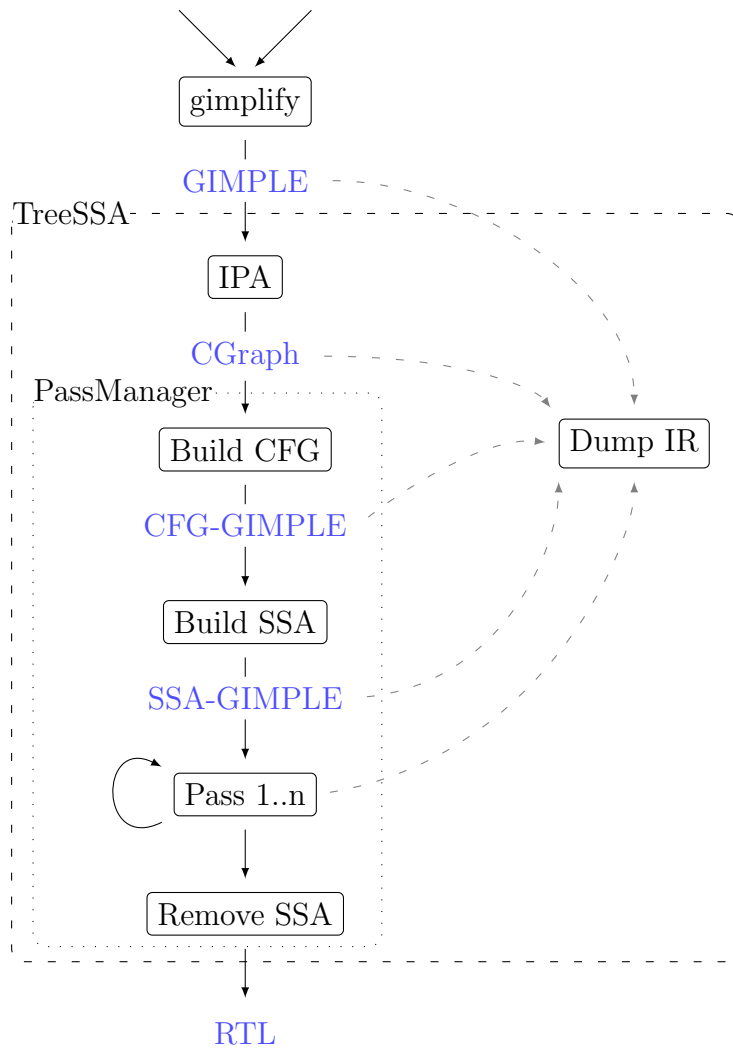


Figure 3.3: Architecture of TreeSSA

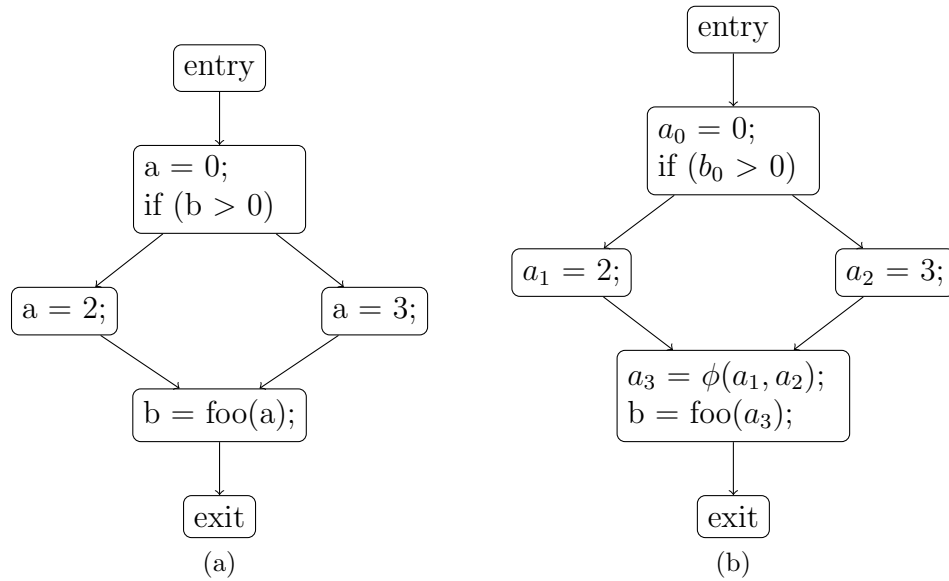


Figure 3.4: Conversion into SSA form

Intermediate representations can be exported at various places as depicted in Figure 3.3. Details about the format of these files can be found in Section 3.3.4 and Section 3.4.

### 3.3.2. Static Single Assignment Form

Computing and maintaining data flow information such as reaching definitions (see Section 2.3.4) is expensive. Static single assignment (SSA) form [CFR<sup>+</sup>91] is a way to make data flow explicit. This representation is used by different optimization passes inside the TreeSSA framework.

A function is in SSA form when each variable is assigned only once. To transform a function into SSA form, every occurrence of a variable  $v$  on the left-hand side of an assignment is replaced with a new version of that variable  $v_n$ . In every point where  $v$  is used it is replaced by the version  $v_n$  that reaches that point. Whenever more than one version may reach a point in the function, i.e. after a branch, the  $\phi$  function is used to create a new version. The semantic of this function is to choose the right version from the set of possible versions.  $\phi$  functions are not computable at compile-time so they have to be removed when all optimizations are finished. An example of a transformation to SSA form is given in Figure 3.3.2. This example also demonstrates the benefits of SSA: it is easy to get all possible definitions for a used variable, which makes e.g. constant propagation (Section 2.1) easier.

GCC uses this approach for variables that always have *killing definitions*, so called *real operands*. A killing definition is an assignment to a variable rendering all previous assignments invalid. Several types of variables have also *may definitions*, i.e. it is not known at compile-time whether a definition changes the variable. These are called *virtual*

### 3. The GNU Compiler Collection

*operands*. The problem is that it is no longer sufficient to link usages with definitions. May definitions also have to be linked with other definitions. The reasons for may definitions are diverse, from aliased and global variables to usage of inline assembler statements.

GCC uses a separate data structure on top of the CFG to implement SSA for virtual operands. This can be exemplified for arrays: It is not feasible to use a versioning scheme for each field inside an array because the field actually used is often not known until runtime. The common approach is to treat each definition of a field inside an array as a may definition of the whole array.

```
1 ...
2 int a[2];
3 # a_2 = V_MAY_DEF <a_1>;
4 a[0] = 0;
5 # a_3 = V_MAY_DEF <a_2>;
6 a[1] = 1;
7 i.0_5 = i_4;
8 # VUSE <a_3>;
9 return a[i];
```

The statement in line 5 indicates that the following statement is a possible definition of *a*. Since the previous definition may not be affected by it, a link to the definition in line 3 is necessary.

Similar approaches are being used by GCC for structures and aliased memory. The problem with may definitions is that each statement possibly has references to a huge number of statements. The current implementation does not scale well, as described in [Nov06b]. An alternative to represent both real and virtual operands in a unified way is described there.

#### 3.3.3. TreeSSA API

The TreeSSA framework offers an infrastructure to access data structures commonly used in optimizers. The following functionality is provided:

- Statement iterators can be used to traverse the body of a function or all statements inside a basic block. Functions to insert or modify statements are available as well.
- Functions to manipulate the CFG or to traverse all basic blocks can be used. Additionally, functions for removing unreachable regions and getting information about loops are available.
- Standard data flow information such as ud- and du-chains is available based on SSA form. Functionality to update these information is provided.

### 3.3.4. Interprocedural Optimizations

The IPA framework, introduced with version 4.1, provides an infrastructure to perform interprocedural optimizations. It generates a call graph for the whole compilation unit, collects information about global variables, and performs inlining based on these data structures.

As discussed in Section 2.3.3, the generation of an accurate call graph is difficult to infeasible when function pointers are used. Method calls in object oriented languages have the same problem as function pointers, and the IPA infrastructure contains neither edges for method calls nor for calls using function pointers.

A textual representation of the call graph can be exported for debugging purposes. It contains information relevant for inlining decisions. Functions are identified by name and information about the exact location of call statements is missing.

## 3.4. Intermediate Representations

Intermediate representations are data structures used to represent the input program inside the compiler. They have a simpler structure that is easier to analyze than the input language. Intermediate representations can be language or target specific, depending on their expected usage. Some optimizations, such as loop transformations, are easier to implement on a representation close to the source code. Lower-level ones, for instance register allocation, work better on a representation close to the target.[HDE<sup>+</sup>92]

Prior to the TreeSSA framework, GCC used only the common intermediate representation RTL. Some front ends shared data structures to represent parse trees, but these were language dependent. Each front end was responsible for the conversion to RTL. The TreeSSA framework introduces two high-level, language and target independent intermediate representations. These three representations are discussed in the following sections. More details about these languages are given in [StGDC05].

### 3.4.1. Register Transfer Language

The Register Transfer Language is a low-level intermediate representation for a virtual machine with an unlimited number of registers. The set of statements is similar to an assembly language: operations to modify memory, compare values, etc. are available. RTL is usually serialized in a LISP-like syntax. A simplified example can be found in Figure 3.5. The if-statement is split up into the comparison in line 2 and a conditional jump in line 7. The return value is set in line 12. Functions are represented as CFGs in RTL optimizers. Methods to manipulate basic blocks and edges are available. Liveness information for registers is also accessible.

Since RTL is a low-level language, it is well-suited for instruction scheduling and code generation in the back end. As mentioned previously, RTL does not contain information about types and accesses to arrays are transformed to offsets to pointers as can be seen in lines 13-14. This can make high-level optimizations difficult.[Nov03] Although RTL is generally machine-independent, details of the representation such as the word-size,

### 3. The GNU Compiler Collection

```
1 int a[2];
2
3 int foo(int i){
4     if(i == 1)
5         return a[1];
6     else
7         return -1;
8 }
```

(a)

```
1 ...
2 (set (reg:CCZ 17 flags)
3     (compare:CCZ
4         (mem/c/i:SI (reg/f:SI 53 virtual-incoming-args) [0 i+0 S4 A32])
5         (const_int 1 [0x1])))
6
7 (if_then_else (ne (reg:CCZ 17 flags)
8     (const_int 0 [0x0]))
9     (label_ref 16)
10    (pc))
11 ...
12 (set (reg:SI 58 [ D.1283 ])
13     (mem/s/j:SI (const:SI (plus:SI (symbol_ref:SI ("a") <var_decl 0xb7b5b108 a>)
14     (const_int 4 [0x4]))) [0 a+4 S4 A32]))
15 ...
```

(b)

Figure 3.5: Simple function inC (a) and corresponding Register Transfer Language (b)

depend on the target. Therefore, the representation of the same function differs slightly when it is compiled for different targets.

### 3.4.2. GENERIC

GENERIC is a high-level language and target independent representation similar to AST. Its goal is to replace RTL as the interface between front end and optimization phase. All language-dependent constructs, such as vTable lookups, are explicit in GENERIC. [Nov06a]

Currently, no optimizations are performed on GENERIC although high-level optimizations are possible. One reason for this is the structural complexity that is allowed in GENERIC. Statements may be arbitrary nested which makes the implementation of optimizers complicated. Furthermore, the C and C++ front ends do not emit GENERIC, so both would not benefit from such optimizers.

The purpose of GENERIC is twofold: First, it provides an easy way to integrate front ends for new input languages. Additionally, it simplifies the translation to the next intermediate representation GIMPLE.

### 3.4.3. GIMPLE

Analyzing a real programming language or a high-level intermediate representation such as GENERIC has advantages and drawbacks. Valuable information, such as type information and array references, is available but the possible complexity of statements makes the implementation of optimizers difficult.

SIMPLE is an intermediate representation developed for the compiler McCAT.[HDE<sup>+</sup>92] It is designed to avoid drawbacks of high-level representations while keeping the advantages. High-level features such as type information and array references are modeled directly in SIMPLE. Complex statements are broken into simpler ones and the flow of control is made explicit. Another problem of real-world languages is the ambiguous semantic. In C, the statement `a = b` may either denote an assignment of a value, if `a` is a scalar, or of an address, if `a` is an array.

GIMPLE was heavily influenced by this intermediate representation and has the following features:

- Expressions, except function calls, can only have 3 operands.
- The only control flow statements are if-then statements.
- Ambiguous semantics are made explicit.

GENERIC or a language dependent tree are transformed to GIMPLE by the so called *gimplifier*. Complex statements are split into GIMPLE compliant statements by introducing temporary variables. Afterwards, the CFG for the current function is generated. simplified and SSA form is generated. The optimizers then work on this SSA form of GIMPLE which is later transformed into RTL.

### 3. The GNU Compiler Collection

```
1 int f(int i){
2     int j = 0;
3     while(i++ < 10){
4         int k;
5         k = bar(i,i+1,j);
6         j += k;
7     }
8     return j;
9 }
```

(a)

```
1 f (i) {
2     int j;
3     int D.5, D.6, D.8;
4     _Bool D.7;
5
6     j = 0;
7     goto <D3>;
8 <D2>; {
9     int k;
10
11     D.5 = i + 1;
12     D.6 = bar (i, D.5, j);
13     k = D.6;
14     j = j + k;
15 }
16 <D3>;
17 D.7 = i <= 9;
18 i = i + 1;
19 if (D.7) {
20     goto <D2>;
21 } else {
22     goto <D4>;
23 }
24 <D4>;
25 D.8 = j;
26 return D.8;
27 }
28
29
30
```

```
1 f (i) {
2     int k, j;
3     int D.5, D.6, D.8;
4     _Bool D.7;
5
6     # BLOCK 0 PRED: ENTRY (fallthru)
7     j = 0;
8     goto <bb 2> (<L1>);
9     # SUCC: 2 (fallthru)
10
11     # BLOCK 1 PRED: 2 (true)
12 <L0>;
13     D.5 = i + 1;
14     D.6 = bar (i, D.5, j);
15     k = D.6;
16     j = j + k;
17     # SUCC: 2 (fallthru)
18
19     # BLOCK 2 PRED: 0,1 (fallthru)
20 <L1>;
21     D.7 = i <= 9;
22     i = i + 1;
23     if (D.7) goto <L0>; else goto <L2>;
24     # SUCC: 1 (true) 3 (false)
25
26     # BLOCK 3 PRED: 2 (false)
27 <L2>;
28     D.8 = j;
29     return D.8;
30     # SUCC: EXIT
31 }
```

(b)

(c)

Figure 3.6: Simple function in C (a) and corresponding GIMPLE (b) and CFG

The example in Figure 3.4.3 demonstrates the conversion in a C-like serialization: the complex statement in line 3 is broken into simpler statements, each addressing at most 3 operands, without implicit side-effects. The loop is transformed into an if-else statement with `gotos`, making the control flow explicit.

The generation of the CFG is simple due to the regular structure. An example can be found in Figure 3.4.3(c). As mentioned in Section 2.3.2, exception handling complicates the generation of a CFG. The TreeSSA framework creates a new basic block for each statement that may throw an exception. This has the advantage that the control flow is explicit with the drawback of a possibly huge number of edges.

#### **3.4.4. Exporting Intermediate Representations**

GCC is able to export each of the previously described intermediate representations at various stages during compilation. This functionality was used to generate the examples in the previous sections. RTL may be exported in a LISP-like syntax as already seen in Figure 3.5. Two different formats may be exported for `GENERIC` and `GIMPLE`. Either the C-like representation used in the examples above or a raw format describing the internal tree structure.

The purpose of these exported files is to allow debugging of the different transformation steps inside the compiler. They are intentionally incomplete to circumvent the development of compiler back ends based on these debugging files.

### 3. *The GNU Compiler Collection*

## 4. Using the TreeSSA framework for security-related Static Analysis

The goal of this thesis is to evaluate whether it is feasible to reuse GCC's intermediate representations for security-related static analysis. The description of intermediate representations inside the GCC already reveals similarities to the data structure requirements discussed in Section 2.4.

The purpose of this chapter is to discuss how static checkers may access GCC's data structures, and which intermediate representation is suitable. First, a motivation of the combination is given Section 4.1. The three intermediate representations described in Section 3.4 are evaluated in Section 4.2 and one of them is selected. An architecture to accomplish the combination is proposed in Section 4.3.

### 4.1. Motivation

Although some static checkers operate directly on the source code, more advanced analyses require a representation of the target program that is easier to analyze, as discussed in Section 2.4. In this case, the first steps of static checkers are the same as the first steps of compiler front ends. Implementing a compiler front end is a major task on its own, so most static checkers use existing front ends for parsing. For example, the commercial version of `xgcc` uses a front end from the Edison Design Group<sup>1</sup>[Eng06] and BOON uses the C Intermediate Language[NMRW02] framework.

Using a real compiler - not only a front end - has several advantages: Most important, a parse tree is usually not sufficient. It has the same complexity as the programming language leading to complex implementations of analyzers. Advanced static checkers do not analyze the sequence of statements, but the possible flow of control and data dependencies inside a program. Data structures to support this are generated during the optimization phase inside a compiler, thus may be reused by a static checker.

Another problem that may arise are compiler-specific language dialects. These are used by many programs and may not be supported by external front ends. Changing the build-process of large software is sometimes a severe problem.[CDW04] This is easier for static checkers that use the same compiler as the usual build process.

An advantage of GCC is its support for multiple programming languages. All of them are translated to the same intermediate representations. This allows to implement language-independent static checkers for security vulnerabilities that are not limited to

---

<sup>1</sup><http://www.edg.com>

one programming language. Even language-specific checkers benefit from this common representation because an infrastructure to analyze the language-independent format could be implemented.

## 4.2. Choosing an Intermediate Representation

As discussed in Section 2.4, tree representations, control flow graphs and call graphs are the dominating data structures necessary for different analysis approaches. Additionally, data flow information is valuable, even for control flow oriented analyses where it may be used to identify false positives.

The three intermediate representations inside the GCC have different purposes, and thus different properties. GENERIC is not an interesting target for security-related static analysis. It is not used by the front ends for C and C++, whereas many security problems are specific to these languages. Furthermore, the structural complexity is the same as in real-world languages. Each static checker has to be able to handle this. Additionally, GENERIC is currently not used for optimizations. Thus, no data structures, except tree representations, are available.

RTL is used for optimizations, and control flow graphs are generated for it. It also has a much simpler structure than a high-level programming language, making it easier to analyze. Security-related static analysis is possible using RTL, as demonstrated by the tool RTLCheck [Lac06]. The main drawbacks of RTL are missing type information, and the representation of arrays, as discussed in Section 3.4.1. Being a low-level language, RTL is well-suited for analyses close to the target machine. On the other hand, it is difficult to implement analyses closer to the source code.

As discussed in Section 3.4.3, GIMPLE is heavily influenced by the SIMPLE intermediate representation. Many of the design criteria for SIMPLE described in [HDE<sup>+</sup>92] are also suitable for security-related static analysis. Explicit array references and type information allow more accurate alias analysis. The simple structure and clear semantics takes the burden from the static checker to understand all corner-cases of a programming language.

Additionally, GIMPLE is available as both a representation similar to a parse tree, and as a control flow graph. Data flow information is available as well as a call graph. This covers most of the requirements described in Section 2.4, which makes GIMPLE a suitable choice for security-related static analysis.

Different levels of analyses benefit from different intermediate representations. The following have been identified:

1. The SSA form of a function is a control flow graph where data flow information is made explicit. This is valuable for advanced analyses that are control-flow sensitive. The data flow information could be used to prune infeasible execution paths. The following code exemplifies this:

```
int foo(int i){
    if(i > 0)
```

```

        ...
    else
        ...
    ...
    if(i > 0)
        ...
    else
        ...
}

```

Without data flow analysis, a control-sensitive approach has to analyze four execution paths, although only two are feasible. Using SSA form, it is easy to determine whether both usages of  $i$  refer to the same definition, thus possibly ignoring two infeasible paths.

2. Control flow oriented approaches where data flow information is ignored, such as MOPS, could operate on the SSA form of a function, but this would make the implementation unnecessarily complicated. A non-SSA CFG-version is more suitable in these cases.
3. For analyses close to the source that are control-flow insensitive, a tree-representation of the whole function is interesting. Lexical tools such as FlawFinder could use this representation to quickly identify vulnerable function calls. Being control-flow insensitive makes the analysis easier and since this representation is very close to the original source code, communicating possible defects to the programmer is easy.
4. The call graph is used in interprocedural approaches to analyze the relationships between different functions, i.e. to detect recursion. Additionally, it may be used to determine the order in which functions are analyzed.

### 4.3. Architecture

As described in Section 3.2, GCC translates one compilation unit at a time. Therefore, the parse tree or the control flow graph of a function from another compilation unit is not available from inside the GCC. To circumvent this drawback, it was decided to export a serialized version of the intermediate representation to hard disk. One or more checking tools are then able to use this intermediate representation and perform their analysis, as depicted in Figure 4.1.

The existing tree-dumpers available inside the TreeSSA framework are intentionally not complete, neither for GIMPLE nor for RTL. Additionally, the C-like serialization of GIMPLE has almost the same parsing issues as regular programming languages. The raw format is sparsely documented and contains only limited information. A possible approach is to extend the existing raw dump infrastructure, but this would require to

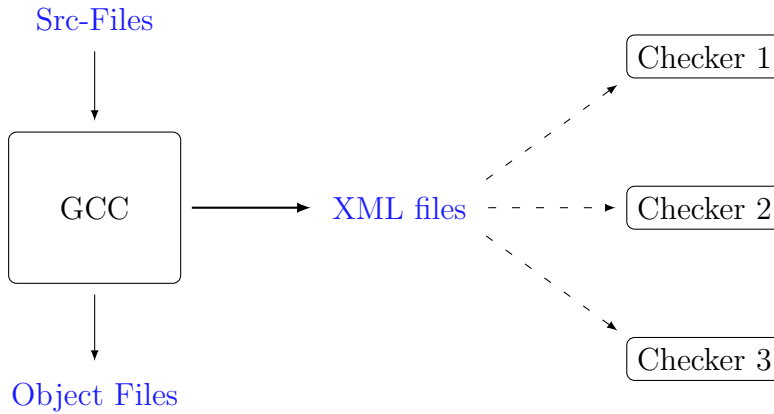


Figure 4.1: Proposed architecture

reverse engineer the current format. Additionally, a parser has to be developed for the static checkers.

Several patches to the GCC allow to export its intermediate representations. GC-CXML [Kit06] and the Node Introspector [DuP06] are patches to the 3.x series of GCC. The former does not export the body of functions, and works only for C and C++. An implementation of the latter for GCC 4.0 has been started, but is still in an early stage. XML-Dump is a patch to GCC version 4.1 that exports GENERIC and the various formats of GIMPLE [Ebn06]. Preliminary tests with this patch where promising, but it is not robust enough to translate complex software. Additionally, many interesting tree-codes, such as information about constructors and inheritance, are not handled by this patch.

Therefore, a new exporting functionality has been implemented as discussed in the next section. Its main goal is to be robust, i.e. able to export complex, real-world software and to contain all information necessary for security-related static analysis, such as type information.

### 4.3.1. On-disk representation

Several compilers have the ability to write their intermediate representation to the hard disk to be able to perform inter-module optimizations, e.g. [Lam06, Dev04]. These formats depend heavily on the internal representations of compiler and translating GIMPLE to one if these is not feasible. Therefore, a data format has been designed, crafted according to the format used inside the GCC. A possible approach is similar to the serialization of RTL. I.e. to use a LISP-like syntax to represent tree relationships. A functional syntax would be well suited to be used in functional languages, which are often used to implement program analyses. For imperative or object-oriented languages, such a syntax is much harder to parse. Additionally a purely LISP-like serialization is not possible, since a lot of information in GIMPLE is internally represented by pointers.

Instead, the Extensible Markup Language (XML) [W3C06b] is used as the external

format. XML is a language-independent, tree-based markup language. Being tree-based, XML is suitable to represent GIMPLE trees efficiently. XML allows to create links between different parts of the document which makes the representation of pointers easier. Additionally, the tool support for XML is mature: several parsers and transformation tools exist. Finally, XML supports the validation of documents against a given XML Schema[W3C06d]. An XML Schema defines the structure of valid documents. This allows to check whether the exported data structure are sensible and complete.

#### 4. *Using the TreeSSA framework for security-related Static Analysis*

## 5. rxd: A robust XML dumper

The goal in developing another method exporting GCC's intermediate representations is to have a *robust* way to write a high-level representation in XML format to the hard disk. This extension is called robust xml dumper (rxd). It is supposed to export data structures required for security-related static analysis for complex, real-world software.

The development of rxd is the topic of this chapter. The integration into the TreeSSA framework is described in Section 5.1. The internal representation of GIMPLE is introduced in Section 5.2. The architecture and implementation to serialize this representation are discussed in Section 5.3. The different formats of the exported data structures are covered in Section 5.4 and an example of rxd's output format is given there. XML Schema is used to validate the output of rxd, as mentioned in Section 4.3.1. The development of the schemas being used is described in Section 5.5.

The source code of rxd and the schemas used for validation can be found on the CD-ROM (see Appendix B).

### 5.1. Integration into TreeSSA

The existing tree-dump infrastructure demonstrates a way how to integrate another dumping facility into the TreeSSA framework. This approach is used for rxd, as depicted in Figure 5.1. Rxd is invoked in the following steps during compilation:

1. The parsing of command-line arguments starts in `tree-dump.c`. The available command-line switches are described in Section 5.3.
2. Exporting the call graph is initiated in `cgraphunit.c`, where the call graph is generated.
3. The gimplifier calls rxd once for each function through the function `dump_function` in `tree-dump.c`. If the GIMPLE format is enabled, rxd appends the function to the XML file.
4. In `passes.c`, rxd is called once for each function after each pass. The generation of the CFG and the transformation to SSA form are also implemented as an optimization passes. Inside rxd, the function is exported when it is enabled for the current pass.
5. After all optimization phases are finished, rxd is called once from `passes.c`. This initiates the export of declarations and types that are used in the translation unit.

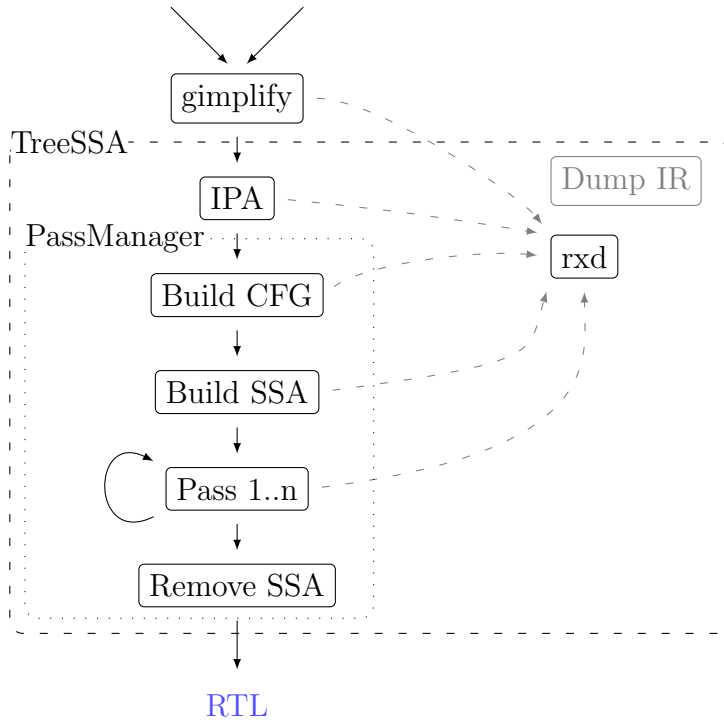


Figure 5.1: Integration into TreeSSA

Most of the changes of existing source files only add function calls and include header files from `rx`d. Modifying GCC's build-process to integrate `rx`d is not a trivial task. XML-Dump (see Section 4.3) provides an example that is adopted for `rx`d.

## 5.2. Internal Representation of GIMPLE

GIMPLE is a tree-based data format to represent the input program inside the TreeSSA framework, as discussed in Section 3.4.3. Each node of a GIMPLE tree has a *tree code*, i.e. an identifier determining the type of this node. 135 different tree codes for GIMPLE are defined in the file `tree.def`. Each tree code belongs to one of ten different *tree code classes*. The class determines the data structure that is used to represent tree codes belonging to this class. For instance, the tree-code `plus_expr` means that this node represent the addition of two values. It belongs to the class `tcc_binary`. The structure for this tree code class contains two pointers to other nodes, which are the operands of the `plus_expr`. The full documentation about the meaning of different tree codes can be found in [StGDC05] or in the files `tree.def` and `tree.h`.

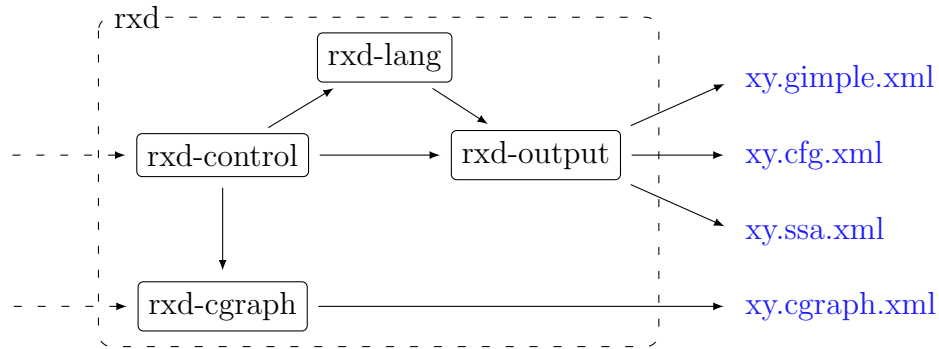


Figure 5.2: Rxd's architecture

Table 5.1: Command-line Switches to control the output format, each has to be prefixed with `-fdump-rxd`

Switch	Filename	Content
<code>-gimple</code>	<code>.gimple.xml</code>	Tree representation of each function
<code>-cfg[-loops]</code>	<code>.cfg.xml</code>	Each function in CFG form, loop information optionally
<code>-ssa[-loops]</code>	<code>.ssa.xml</code>	Each function in SSA form, loop information optionally
<code>-cgraph</code>	<code>.cgraph.xml</code>	Call Graph for the current translation unit
<code>-all[-loops]</code>	<code>*.xml</code>	Enable all of the previous formats, loop information is optionally enabled for CFG and SSA

## 5.3. Implementation

Rxd is not targeting one specific static checker. Instead, it is supposed to provide a general mechanism to access interesting data structures for program analysis. Therefore, the implementation is split into different modules to facilitate modifications necessary for future analyses. Rxd's architecture is depicted in Figure 5.2. Details about the different modules are provided in the following sections.

### 5.3.1. rxd-control

The main module of rxd is `rx-d-control`. Its tasks include the parsing of command-line arguments, traversing the body of functions and to collect information about declarations and types that are used in each translation unit. Rxd allows to export four different formats of a translation unit: a tree representation, a control flow graph, a control flow graph in SSA form and a call graph. The rationale for this selection is given in Section 4.2. Each format may be enabled separately by the command-line switches given in Table 5.1.

After each optimization pass, rxd is called with an identifier for the pass and the function declaration as its arguments. When exporting is enabled for this pass, the function declaration is appended to the corresponding XML file. A detailed description of the output format is given in Section 5.4.2.

## 5. *rx*d: A robust XML dumper

Rxd-control initiates the output of all declarations for the current function, such as arguments and local variables. Afterwards, the body of the function is traversed. If the function is available as a control flow graph, rxd-control iterates over all basic blocks, exporting information about these, the edges for this basic block, and all statements in the body of each basic block. The same function that traverses the body of basic blocks is used to traverse the body of a function that is not available as a control flow graph.

This function, `rxwalk_tree`, uses rxd-output to export information about a GIMPLE node. It then recursively calls itself for each operand of the current node. This function is crafted according to the existing tree dumper in `tree-dump.c`.

The main problem while serializing GIMPLE is its usage of pointers. Whenever a GIMPLE node has a variable declaration as its operand, a pointer to the original variable declaration is used. Types are referenced in the same way, i.e. the type of a statement is a pointer to a type definition. Each type and each declaration has a unique identifier assigned by the front end. Rxd uses these identifiers to serialize pointers referencing types and declarations. Further details about the linking between the use and the declaration of variables and types can be found in Section 5.4.1. Two different approaches are used to export all types and declarations that are referenced inside a translation unit.

Three types of variable declarations are distinguished: global, function and local variables. All global variables are available as a list inside the call graph infrastructure. This list is exported after all functions have been exported by rxd. Function variables are exported together with the arguments of a function. Finally local variables are declared inside `bind_exprs` and are exported as one of their child elements.

The TreeSSA framework does not maintain a list of types that are used inside a translation unit. Therefore, a hash table is used to collect all types that are referenced. After exporting all functions, this table is written to the XML files. Namespaces are handled in the same way, except that a linked list is used instead of a hash table.

The call graph infrastructure offers a list of functions that are used inside a translation unit. During the evaluation (see Section 6.2), it was detected that some functions were missing in this list. Therefore, a hash table is used in the same way it is used for types to collect information about referenced functions.

The TreeSSA framework provides a loop analysis infrastructure that can be used to identify basic blocks inside control flow graphs that form loops. Exporting loop information may be enabled for the CFG and SSA based dump, as shown in Table 5.1. After the infrastructure has analyzed the control flow graph, rxd exports this information in the format described in Section 5.4.2.

### 5.3.2. *rx*d-output

The module rxd-output encapsulates all output to the XML files. It controls the attributes that are exported for each GIMPLE node, such as location information or unique identifiers. Additionally, the attributes for basic blocks, edges and loops are defined here.

An XML library to create the output format is not used for two reasons. First, it would require to integrate a third-party library into the complex build system of GCC.

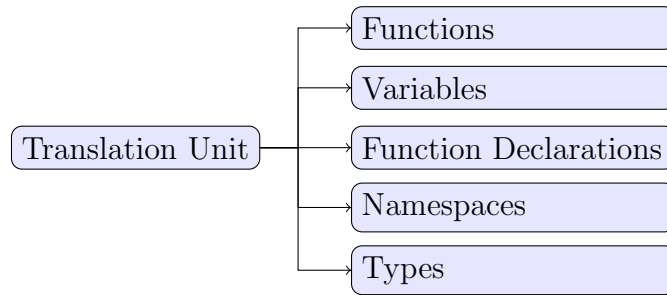


Figure 5.3: Format for translation units

Additionally, GCC uses its own memory management, which might cause problems that are hard to debug. The XML Schema support, described in Section 5.5 is used instead to validate that the output of this module is valid XML.

### 5.3.3. rxd-languagespecific

Although GIMPLE is language-independent, some information about types is held in language-specific structures. A language-specific module is needed to export these data structures. A preliminary version for C++ type information has been implemented. Currently this version only exports vTable information in class definitions.

### 5.3.4. rxd-cgraph

The rxd-cgraph module is called from the IPA infrastructure to export a call graph for the current compilation unit. The call graph consists of nodes and edges, where nodes represent functions and edges represent call statements. The implementation of rxd-cgraph is trivial. Information about each node in the call graph is exported, together with information about all its outgoing edges. The way how edges are linked to call statements inside the GIMPLE representation is described in Section 5.4.3. The call graph is unfortunately not always complete, as mentioned in Section 5.3.1. An investigation of the problem has not yet been performed.

## 5.4. Data Formats

The purpose of this section is to provide a detailed description of the data formats exported by rxd. For each dump format, each compilation unit is exported to its own XML file. The following sections first describe the format of the three different GIMPLE formats. Afterwards, the call graph is discussed in Section 5.4.3.

The overall format is depicted in Figure 5.4, it describes the elements used to structure the representation of translation units. Global information about types, function declarations, namespaces and global variables are grouped together using the appropriate element. Each of these elements has only GIMPLE nodes as child elements. Details

Table 5.2: Attributes used for declarations

Attribute	Meaning
uid	Unique identifier
name	Name of this declaration, if available
typeref	Reference to the type of this declaration
nsref	Reference to the namespace of this declaration

about the representation of GIMPLE in XML can be found in the next section. Function definitions use a more complex format which is described in Section 5.4.2.

An example of a control-flow graph produced by *rx*d and the corresponding call graph for the simple example in Figure 5.5 can be found in Figure 5.6 and Figure 5.8. These are used to exemplify the important concepts in the following sections.

### 5.4.1. GIMPLE Statements

Each node in a GIMPLE tree is represented as an element inside the XML hierarchy. Operands of nodes are represented as child elements. Each node has different attributes, depending on the class it belongs to. The following classes are distinguished:

**tcc\_constant** All tree-codes in this class represent constant values, such as integer values or strings. Each of them has its value as its child element. Attributes are not used. An example of an integer value can be found in Figure 5.6 in line 16.

**tcc\_type** The information about each node representing a type is different for each kind of type. GIMPLE defines 17 different basic types, e.g. for integers, pointers, or aggregate types that are similar to structures in C. Each type has its unique identifier as an attribute. Whenever a GIMPLE node use a type, this identifier is used to determine the type definition. Aggregate types use child elements to represent their members, and function types use child elements to represent argument and return types. This can be found in the example starting in line 36. The type of the arguments is determined by the attribute `typeref`, e.g. the function references an integer type and a void type. Scalar types have attributes depending on the kind of type they represent, an example of an integer type can be found in line 33.

**tcc\_declaration** Declarations are used in two different ways: either they represent the usage of a declaration, or the declaration itself is meant. Only two attributes are used in the former case, that is a unique identifier to have a reference to the real declaration and the name of the variable<sup>1</sup>. The attributes of real declarations and their meaning can be found in Table 5.2. Additionally, variable and function declarations contain flags to determine their visibility, as shown in Figure 5.6 in line 28 and 29. The function called in line 13 uses the attribute `ref` to determine that it references the function declaration in line 28.

<sup>1</sup>The name of the variable is redundant here, but helps a lot to understand the exported files.

**tcc\_expression and co.** Six tree code classes are represented by the same data structure. The main difference between these classes is the number of operands, e.g. `tcc_binary` has exactly two. Two attributes are used for this class of nodes: the location, if available, contains a file name and a line number. Nodes that represent type conversions also have a reference to a type as an attribute.

`Call_expr`, i.e. nodes representing function calls, also have an attribute containing their memory address inside the GCC. This is used by the call graph to identify the exact function call, as described in Section 5.4.3.

**tcc\_exceptional** Each tree code in this class uses a different structure, thus the XML representation is also different. A broad range of tree codes belongs to this class, such as `tree_list` which is used to represent arguments to function calls (see line 15). A full definition of the exported formats can be found in the XML Schema described in Section 5.5.

## 5.4.2. Function Definitions

Rxd's format used for function definitions can be found in Figure 5.4.2. Each function definition has a reference to its declaration and a number of child elements. Parameters are declared inside the arguments element, and static chain may contain a reference to one of these parameters. This is used by GCC to implement nested functions. Local variables may be declared inside the declarations element, while the variable that contains the return value is declared in result.

Two different formats are used for the body of a function, depending on the desired output format.

### Tree representation

During the first phases of optimization, functions are available as an abstract syntax tree. A `bind_expr` creates the scope for a function, i.e. local variables are declared there. The statements of the function are contained in a `statement_list`. Each statement inside this list is a GIMPLE node in the representation described above.

### Control Flow Graph

The CFG and SSA format both use a control flow graph to represent the body of functions. Each basic block has its unique identifier and its loop depth as its number. The loop depth determines how many loops are enclosing this basic block. The child element `statement_list` is used to represent all statements in the body of the basic block. PHI-nodes used in the SSA-version are also represented as child elements.

Edges are used to connect basic blocks by their unique id. Flags are used to describe the kind of edge, for instance whether it is taken on true or false. Loops are using the basic block index to determine the header and latch of the loop.

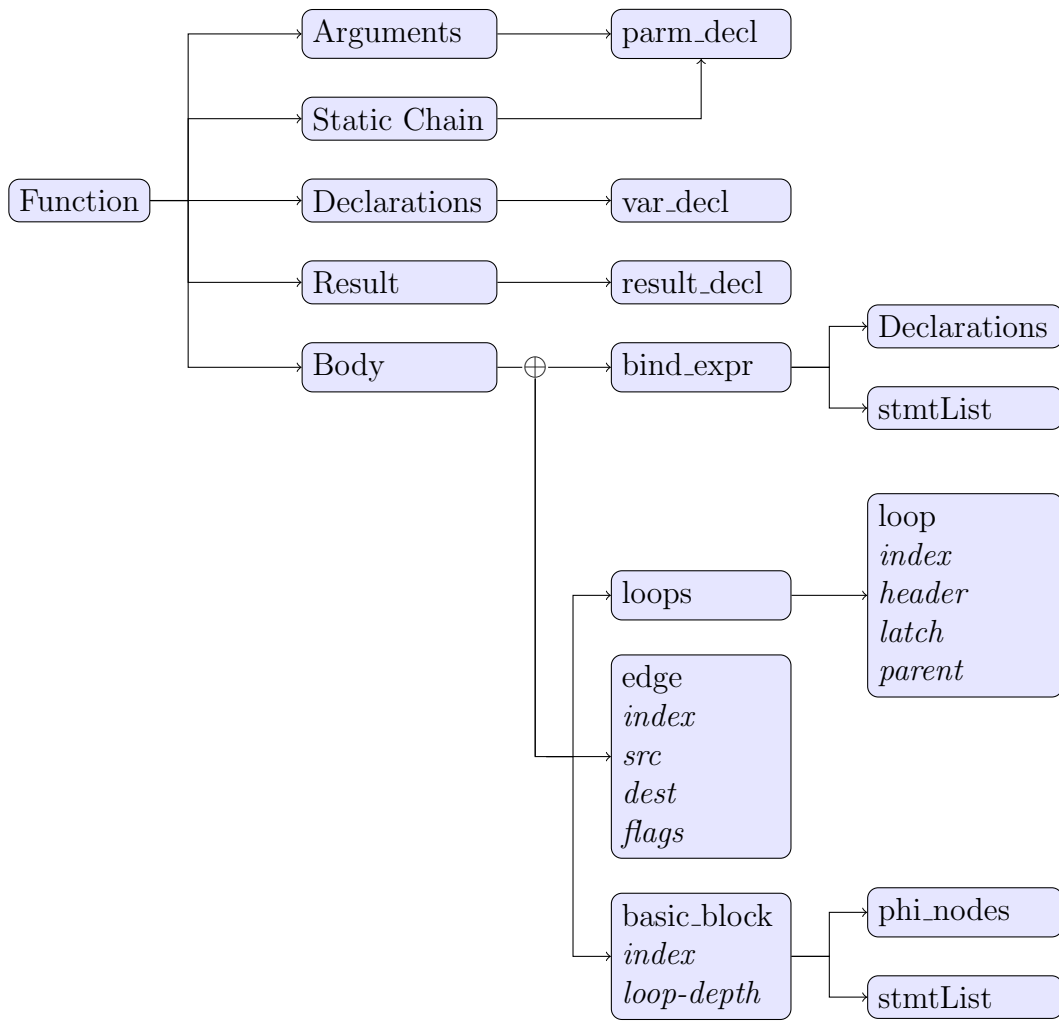


Figure 5.4: Format for functions

```

1 int bar(int);
2
3 void foo(void){
4     bar(1);
5 }
    
```

Figure 5.5: Source code used in following examples

```

1 <translation_unit xmlns:xsi=...>
2 <functions>
3   <function_decl ref="D2" name="foo">
4     <result>
5       <result_decl uid="D3" typeref="T25" location="ge.c:3" />
6     </result>
7     <body>
8       <basic_block index="D2-BB-1"/>
9       <edge src="D2-BB-1" dest="D2-BB0" fall-thru="true" />
10      <basic_block index="D2-BB0" >
11        <call_expr address="0xb7c03028" location="ge.c:4">
12          <addr_expr>
13            <function_decl ref="D1" name="bar"/>
14          </addr_expr>
15          <tree_list>
16            <integer_cst>1</integer_cst>
17          </tree_list>
18        </call_expr>
19        <return_expr location="ge.c:5">
20        </return_expr>
21      </basic_block>
22      <edge src="D2-BB0" dest="D2-BB-2" />
23      <basic_block index="D2-BB-2"/>
24    </body>
25  </function_decl>
26 </functions>
27 <function_declarations>
28   <function_decl name="bar" uid="D1" typeref="T75" location="ge.c:1"
29     external="true" public="true" />
30   <function_decl name="foo" uid="D2" typeref="T54" location="ge.c:3" static="true" />
31 </function_declarations>
32 <types>
33   <integer_type typeuid="T8" name="int" size="32" precision="32"
34     min="-2147483648" max="2147483647"/>
35   <void_type typeuid="T25" name="void"/>
36   <function_type typeuid="T75">
37     <arguments>
38       <arg typeref="T8" />
39       <arg typeref="T25" />
40     </arguments>
41     <return typeref="T8" />
42   </function_type>
43   <function_type typeuid="T54">
44     <arguments>
45       <arg typeref="T25" />
46     </arguments>
47     <return typeref="T25" />
48   </function_type>
49 </types>
50 </translation_unit>

```

Figure 5.6: Example of rxd's control flow format

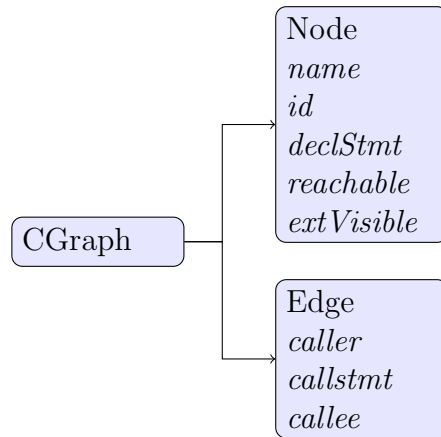


Figure 5.7: Format for call graphs

```

1 <?xml version="1.0"?>
2 <cgraph xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
3   xsi:noNamespaceSchemaLocation='cgraph.xsd'>
4   <node name="bar" id="N1" declstmt="D1277" reachable="0" extVisible="0"/>
5   <node name="foo" id="N0" declstmt="D1279" reachable="1" extVisible="0"/>
6   <edge caller="N0" callstmt="0xb7b82028" callee="N1"/>
7 </cgraph>
  
```

Figure 5.8: Example of Rxd's call graph format

### 5.4.3. Call Graphs

The call graph is exported to a separate file because it can be used for all of the three different output formats. The call graph contains nodes which represent function declarations and edges that represent call statements. Each node has information about its visibility, the reachability and a unique identifier to identify the function declaration that it represents.

Edges in the call graph contain pointers to the caller and callee node which can be serialized using the unique node identifier. Additionally, a pointer identifies the call statement that is responsible for this edge. Call statements do not have unique identifiers, hence another way to serialize the pointer is necessary. In the current implementation, the memory address of the call statement is used as an identifier. The edge in Figure 5.7 references the call statement in Figure 5.6 line 11. The memory address is changing for each invocation of the compiler, so this linking only works between files that are created during one compilation. The only alternative would be to modify each front end inside the GCC, which is not feasible.

## 5.5. Validation

. The implementation of rxd is not straight forward, for instance there is no way to get access to all types available inside one compilation unit. XML Schema is used to validate that all relevant information of the input program is exported to the XML files.

XML Schemas

...provide a means for defining the structure, content and semantics of XML documents. [W3C06d]

XML documents can be validated against a given schema to ensure they have the correct format. XML Schema is a very powerful language supporting inheritance as in object oriented programming languages.

Two schemas to validate the output of rxd are used, one for exported translation units and one for call graphs. These schemas describe valid data format, as defined in the previous section. Both of them can be found on the CD-ROM (see Appendix B).

The XML Schema for whole translation units enforces the general structure of the exported files as described in Section 5.4. Two additional properties are ensured: The Rough GIMPLE Grammar [StGDC05] has been implemented to ensure that each statement has only valid operands. As indicated by the name of this grammar, it does not describe all valid GIMPLE statements. Therefore, the schema has been iteratively extended using small test cases.

The most important point in using XML Schema is the automatic checking of key-references relationships. Inside an XML schema, an attribute can be defined as an *id*, and other attributes can be defined as *idref*. When a document is validated, the XML Schema validator ensures that each identifier is unique, and each referenced identifier exists. It is used to validate that each referenced variable is actually declared exactly once, and that only defined types are referenced.

The structure of call graphs is simple, which makes the implementation of a corresponding XML Schema straight forward. The main contribution of this schema is the automatic checking of references between nodes and edges in the call graph.

5. *rxcd: A robust XML dumper*

## 6. Discussion

The extension to the TreeSSA framework, as motivated in Chapter 4 and described in Chapter 5 is evaluated in this chapter. A proof of concept analysis has been developed in order to evaluate the usability of the exported data structures for program analyses. This implementation is the topic of Section 6.1. The usability of rxd regarding real-world software is evaluated in Section 6.2. The limitations the approach in general and rxd specifically are discussed in Section 6.3, while related work is outlined in Section 6.4.

### 6.1. Proof of Concept Analysis

The purpose of the proof of concept analysis is to demonstrate the usability of the exported intermediate representations for security-related static analysis. The development of a new approach to detect security-critical errors is not feasible due to time constraints. An alternative would be to reimplement a well-known static checker. The only tool where this possible within a reasonable time frame is FlawFinder. This approach would be able to detect vulnerabilities in real-world software, namely the same as the original tool. However, the main contribution of FlawFinder is its database of vulnerable function calls. The only data structure that could be used is a tree representation.

Instead, the algorithm described in Section 2.2 is chosen as the proof of concept analysis. Although it is not designed to detect security vulnerabilities in real-world software, it may serve as an example for program analysis in general. The proof of concept analysis targets C programs, and two extensions to the original algorithm were made. In order to get more accurate results, the alternative widening operator proposed by Nielson [NNH99] is used (see Section 2.2).

The original algorithm is designed for a language without functions. Two changes are necessary to analyze languages supporting functions. The original algorithm initializes the interval for each integer variable on all edges with the empty interval. The modified version treats the first edge in the flow differently. The intervals for all integer arguments to the current function are initialized with the largest possible value, i.e.  $[-\infty, \infty]$ . Additionally, a way to handle function calls is necessary. In the first version, function calls always return the largest possible interval. A refinement is described in Section 6.1.3.

#### 6.1.1. Requirements

As an input, the algorithm expects the program in the following form:

## 6. Discussion

1. Programs are represented as flow graphs where nodes represent single statements.
2. Headers of loops are already identified in the flow graph.
3. Nodes containing conditional statements have no side-effects.

Additionally, the extended widening operator requires a set of all integral values inside a function.

The data format exported by rxd fulfils these requirements, except two minor differences. Nodes in the flow graph exported by rxd represent basic blocks and not single statements, but a transformation is trivial. A list of integer variables is easily accessible, as described in the next section.

### 6.1.2. Implementation

The algorithm has been implemented in Java using the Xerces XML parser<sup>1</sup> to access XML files. The translation of the algorithm to Java is straight forward, once the infrastructure for interval arithmetic and access to the control flow graph are available. Classes to represent intervals and contexts have been implemented, although only a limited number of operations is supported. Two details of the implementation are important for this evaluation: the access to XML documents and the representation of GIMPLE statements.

The Document Object Model (DOM) [W3C06a] provides an object-oriented interface to XML documents. Each translation unit is represented as a DOM document, and elements inside this document are addressed using XPath [W3C06c]. XPath is a language to address elements inside XML documents similar to path names in UNIX. For example, the XPath expression `translation_unit/types/*` returns a list of all type definitions in the current translation unit, and `//integer_cst` returns a list of all integers constants.

A class hierarchy is used to represent GIMPLE statements. Although all the operations could be performed directly on XML elements, it simplifies the implementation of the algorithm significantly. Each class in this hierarchy implements the method `interpret`. This method creates the mapping between GIMPLE expressions and interval operations. E.g. a `plus_expr` is represented by an object of type `tcc.binary`. The `interpret` method of this class creates a new interval by adding the intervals of its operands. The implementation of this hierarchy is simplified by the regular structure of GIMPLE statements. A `plus_expr` has always exactly two operands, both either a constant value or a variable reference. Another advantage is that GIMPLE expressions are side-effect free. Therefore, it is not necessary to e.g. evaluate all arguments of a function call.

### 6.1.3. Intermodule Extension

An extension to the intraprocedural analysis has been implemented to evaluate how analyses may incorporate several translation units. An interprocedural extension to

---

<sup>1</sup><http://xerces.apache.org/xerces2-j/>

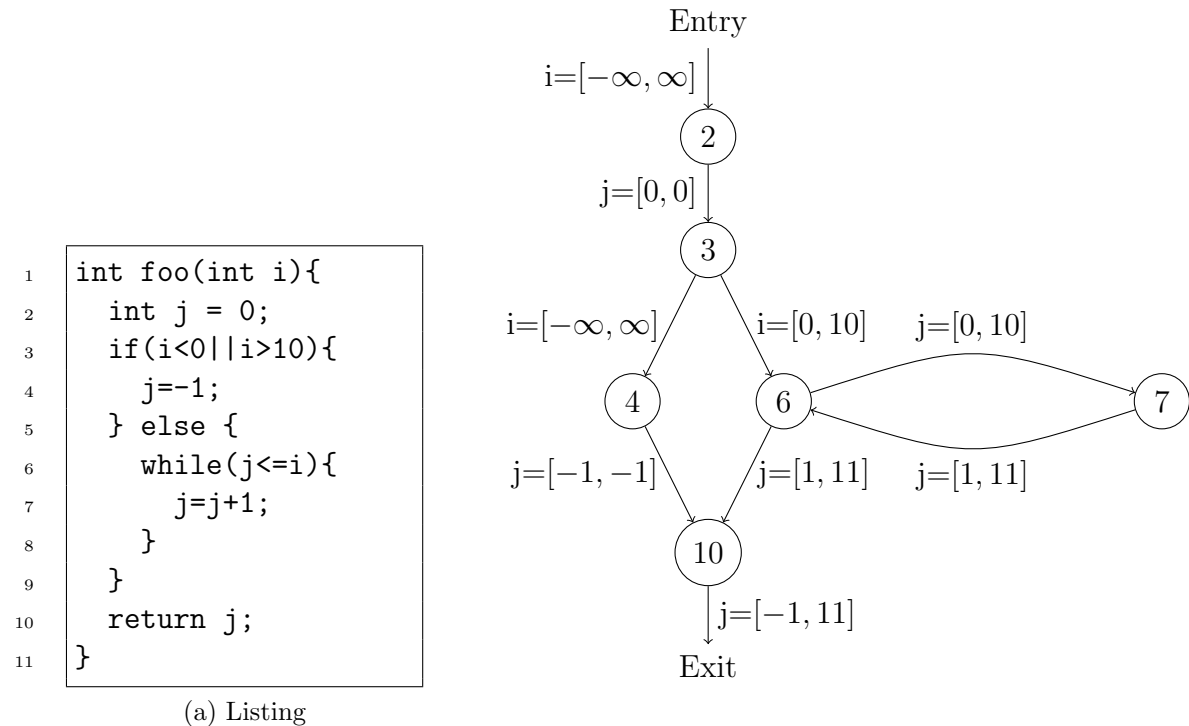


Figure 6.1: Result of an integer range analysis

the original algorithm is described in [CC77b]. Although it is an interesting analysis, an implementation was not feasible due to time constraints. Instead, a worst-case extension in the sense of [CC02] has been defined. The idea is to ignore the context of a function call, and instead analyze all functions on their own. The results gained from these analyses are then combined to get an analysis for the whole program.

This worst-case analysis makes no assumptions about the context in which a function is called, so all function arguments are set to the largest interval  $[-\infty, \infty]$ . The analysis starts with the `main` function and applies the intraprocedural algorithm to it. Whenever a function call occurs, the analysis of the caller is suspended and the callee is analyzed. After finishing the analysis of the callee, the determined return value is used in the analysis of the caller. If functions call each other recursively, the worst case return value is assumed.

#### 6.1.4. Results

The result of the analysis of a simple function is depicted in Figure 6.1. The source code of the proof of concept analysis, the XML representation of the function and the automatically generated graph used to create Figure 6.1 can be found on the CD-ROM (see Appendix B).

The results gained from such an analysis are, except for carefully crafted examples, not satisfying. The abstraction by intervals is too coarse and does not represent relationships

between variables. Nothing is known about arguments of a function, because the context of function calls is ignored.

Nevertheless, the goal of this proof of concept is to demonstrate the usability of the exported data structures. Accessing XML using Xerces is easy, and the navigation inside XML documents is comfortable using XPath. GIMPLE's regular structure facilitated the implementation of the class hierarchy significantly, as described above. The extended analysis described in Section 6.1.3 demonstrates that intermodule analysis is feasible using rxd's output format. However, a limitation for this class of analyses is discussed in Section 6.3.5.

## 6.2. Practical Evaluation

This work focuses on practical static analysis applicable to real-world software. The usability of rxd regarding this class of programs is evaluated in the following sections. First, the software packages being used are introduced. Afterwards, the compilation speed and the size of the exported data structures is evaluated. Finally, the integration of rxd into the build process and its robustness are discussed.

The evaluation was performed on an AMD Athlon XP 2600+ with 1024 MB RAM using Linux 2.6.15. The Sun Java 1.5.0 JDK was used.

### 6.2.1. rxd and real-world software

Three software packages with different characteristics are used for the following evaluation. The GNU C library (glibc) [Fre06b] is the standard C library for the GNU system and most Linux systems. The glibc is a large piece of software using many advanced compiler specific features of GCC, such as exception handling and nested functions. Additionally, it is an interesting target to evaluate the robustness of rxd. Many security vulnerabilities are based on the wrong usage of functions, such as `strcat` that also are provided by glibc. An easily accessible format of these functions is useful to detect this kind of vulnerabilities.

Sendmail [Sen06] is a commonly used mail transfer agent written in C. Due to its long history of security vulnerabilities, it is one of the commonly used examples to evaluate new static checkers, e.g. in [WFBA00, CW02].

Heatload [Poe06] is used to evaluate rxd regarding its usability for object oriented languages. Heatload is a graphical program written in C++. Although the implementation itself is small, a large GUI framework is used.

### 6.2.2. Compilation Time

Rxd is called in several steps during the compilation, and the generated XML files are large (see Section 6.2.3). Therefore, an evaluation of the increased compilation time has been performed. Each package has been built four times using GCC without the rxd extension, and four times with rxd's CFG export enabled. The results can be found in

Table 6.1: Average compilation Time with and without rxd

	without rxd	rxd enabled	$\delta$
glibc	22:22.45 min	22:47.60 min	101.87 %
Sendmail	1:02.41 min	1:03.87 min	102.35 %
Heatload	0:03.72 min	0:03.19 min	116.54%

Table 6.2: Space requirements for rxd output files

	Overall	Source Code	Source Code + Library headers	rxd CFG format
glibc	115 MB	38MB	38 MB	370 MB
sendmail	8.4 MB	3.3 MB	3.9 MB	35 MB
heatload	400 KB	9.27 KB	2.5 MB	3,3 MB

Table 6.1. The script used to generate this data and the numbers from all iterations are provided on the CD-ROM (see Appendix B).

Both relative and absolute increase for glibc and Sendmail are low. Hence it is feasible to have rxd enabled during the whole development cycle. The higher relative increase for Heatload stems from the short absolute time and the huge resulting file size, as discussed in the next section.

### 6.2.3. Space

Tree-representations of programs are usually significantly larger than the source code itself. [Wad90] A major concern when serializing tree representations is thus the resulting file size. Additionally, XML is known to be verbose. Therefore, an evaluation of rxd's output files regarding their file size has been performed.

Rxd is used to export the CFG format for each of the three evaluation targets. All source and header files that are compiled during the build process are recorded. The resulting file sizes can be found in Table 6.2. The exported CFG files for glibc and Sendmail are roughly ten times larger than the corresponding source code. Rxd's output for Heatload is roughly 340 times larger than the source code. This is due to the number of header files that are included. 2.5 MB of header files from the GUI framework are included in Heatload.

The exported intermediate representations are significantly larger than the corresponding source files. This is especially a problem for graphical tools, as shown by Heatload. Storing this amount of data is not a problem, since nowadays hard discs are large enough even for much bigger programs. Nevertheless, parsing of these data structures has to be fast enough to be usable inside static checkers.

Table 6.3: Average time to validate rxd's output

	Time in minutes
glibc	02:37,21
Sendmail	00:15:12

### 6.2.4. Validation

The validation of the exported data structures has two purposes. It demonstrates that rxd's exported intermediate representations are complete, and can be used to evaluate the parsing time. In order to be able to use XML documents containing key-relationships (see Section 5.5), the document has to be parsed by a validating parser. Hence the validation time measured in this section may serve as an estimation of the parsing time.

Rxd's output for glibc and Sendmail has been validated against the XML Schema described in Section 5.5. A simple Java program using Xerces is used as the validator, this tool can be found on the CD-ROM (see Appendix B). The results can be found in Table 6.3. The validation time for 370 MB of XML files is remarkably low. So although the exported intermediate representations are large, their parsing is fast enough to be usable for static checkers.

The output for Heatload could not be validated, the reasons are discussed in Section 6.3.

### 6.2.5. Integration into the Build-process

The build process of large software packages is often complex, as many dependencies between different modules have to be obeyed. Problems with different approaches of integration and possible solutions are described in [CDW04]. The three examples used during evaluation demonstrate how rxd can be integrated into build processes.

Glibc and Heatload use Autotools<sup>2</sup> as their build system. Two environment variables are sufficient to export GCC's intermediate representations. Many other open source packages also use this build system, so the integration of rxd is similar. Sendmail uses a different build system, although the integration is also simple. The detailed steps to enable rxd for the three packages can be found in the shell scripts used in previous evaluations (see Appendix B).

### 6.2.6. Robustness

Three software packages with different characteristics have been used to evaluate rxd. All code was successfully compiled, and the output for glibc and Sendmail has been validated. Having evaluated two large and complex C packages provides confidence that all C code is processable using rxd. Only a small amount of C++ code has been

<sup>2</sup><http://www.gnu.org/software/autoconf/>

evaluated, and some tests with Java and FORTRAN where promising, but a thorough evaluation is still necessary.

## 6.3. Limitations

Although GIMPLE is an interesting choice for security-related static analysis, the proposed solution has some drawbacks and limitations. Rxd's current limitations are the topic of Section 6.3.1 and Section 6.3.2. Problems with object-oriented languages are discussed in Section 6.3.3. Limitations regarding the integration of annotations are outlined in Section 6.3.4, while problems arising through intermodule analysis are discussed in Section 6.3.5.

### 6.3.1. Implementation Limitations

In the current implementation, only scalar variables are exported in SSA form. The SSA form for virtual operands is stored in a separate data structure which is difficult to serialize. Additionally, the virtual SSA form is regarded as almost too large for the use inside the GCC, which would create even larger XML files.

The incomplete call graph is a severe drawback for rxd-cgraph. A detailed analysis of the functions that are not collected by the IPA infrastructure would be necessary to solve this problem. An alternative approach is to port rxd to the current development version of GCC in order to evaluate whether the problem is still existing.

### 6.3.2. Unsupported Tree Codes

Currently, 96 tree codes out of 135 possible GIMPLE tree codes are covered by the XML Schema and tested either by test cases or the code used during evaluation. 34 tree codes are classified as trivial and ignored because they are used only for FORTRAN or Ada programs. An example are the various division operators defined for FORTRAN. These tree codes are exported by rxd, but can not be validated for now. One tree code is ignored by the current implementation because it is only used for Ada programs. Four tree codes are not handled at the moment because no test case has yet been found. A list of all tree codes with their implementation status can be found on the CD-ROM (see Appendix B).

### 6.3.3. Object-Oriented Languages

GIMPLE is not object oriented, thus object-oriented features such as method invocations are transformed to imperative equivalents during the translation to GIMPLE. All semantics of a method call has to be explicit in GIMPLE. This can be exemplified for a method call in Java and its translation to GIMPLE.

In Figure 6.2, a simple method call in Java and its corresponding GIMPLE representation can be found. Figure 6.2(a) depicts the AST of a method call inside the Eclipse

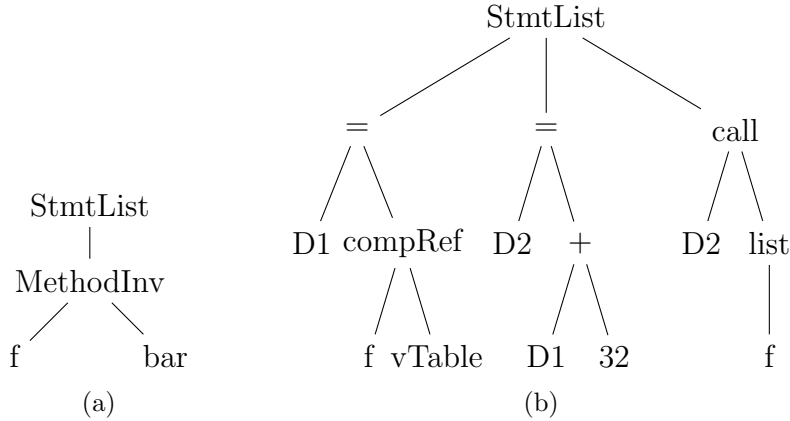


Figure 6.2: Method invocation in Java (a) and GIMPLE (b)

Java Compiler [The06]. A simplified version<sup>3</sup> of the equivalent in GIMPLE is given in Figure 6.2(b). It works as follows: First, a reference to the `vTable` is acquired. The `vTable` contains information about the methods available for an object, depending on its type. `32` is added to the pointer to the `vTable`, where `32` is the `vTable` index determining the method that is actually called. Finally, the function is called using the previously constructed pointer, with a reference to the object as its first argument. This representation of method calls is much harder to analyze than the direct representation inside Eclipse. Other features, such as object creation, are even more complex. Possible solutions to this problem are discussed in Section 7.1.1.

Another limitation of `rxid` regarding object-oriented languages are variables that are used, but not declared somewhere in the GIMPLE tree. The Java front end uses static variables during constructor calls to determine the class of the object that is created. These variables are not declared somewhere inside the GIMPLE tree. This results in a validation error, because a variable that is used has not been declared. A similar problem exist for special functions in C++. This is one reason why `rxid`'s output for Heatload can not be validated.

Another problem arises when a C++ program uses templates. Tree codes to represent template information are used inside GIMPLE trees, although they are not valid GIMPLE. This is not documented inside the description of GIMPLE, therefore it is currently not supported by the XML Schema. This is the other reason why the `rxid` output for Heatload can not be validated.

### 6.3.4. Annotations

Annotations are meta data inside programs that provide additional information about the function. Modern languages, such as Java, support annotations directly. They are used e.g. to generate interfaces or to execute tests automatically. Annotations are also

<sup>3</sup>Two type cast have been removed.

used by static checkers, such as Splint [EL02] to allow the programmer to specify pre- and postconditions for functions.

Integrating annotations into a language without modifying the parser is not trivial. On the one hand, one might use specially named variables inside procedures to carry additional information that can be used in the analysis. A string variable with name `preconditions` could be used to specify the preconditions of a function.

Another feasible approach using `rxd` is to add annotations as comments to the source code. After the source files have been exported as XML, a tool could add the annotations to the exported files.

### 6.3.5. Intermodule Analysis

Whenever a static checker analyzes more than one translation unit, it is confronted with the same problems as a linker. I.e. there may be two functions or variables with the same name in two different compilation units. Usually, GCC is called with a list of source or object files that are supposed to be linked together. Unfortunately, this information is not available when `rxd` is executed.

The approach taken in the proof of concept analysis is to explicitly name all translation units that are supposed to be analyzed. This is not practical for large software packages. A feasible approach is to analyze Makefiles to acquire the missing information.

## 6.4. Related Work

RTLCheck is a framework that performs abstract interpretation on RTL.[Lac06] The intermediate representation RTL is exported to the hard disk and used by an abstract interpreter. Since RTL is a low-level representation, the implemented analyses are also targeting low-level errors, such as integer overflow analysis.[Mue06] The analyses currently implemented by RTLCheck are feasible using `rxd`'s exported data structures, although a thorough comparison is yet missing.

Two approaches, `myGCC` and `GEM`, support the integration of static checkers into the TreeSSA framework. `myGCC` allows the specification of syntax patterns to detect common programming problems. `GEM` allows to dynamically load libraries into the GCC and perform checks on a GIMPLE representation. Since both approaches are executed inside the GCC, only intraprocedural analysis is possible.

The C Intermediate Language (CIL) is an “Infrastructure for C Program Analysis and Transformation”.[NMRW02] It transforms C into a simpler, more explicit intermediate representation, similar to the transformations during the gimplification. Modules to merge different compilation units, to generate a CFG, and to perform standard data flow analysis are available. CIL is used by BOON (see Section 2.4.2), but has the drawback that it only supports one input language.

The SUIF2 compiler infrastructure [Lam06] defines a language and target independent intermediate representation. SUIF2 does not include language front ends, external front ends are used instead. The intermediate representation allows high-level analysis, and

## 6. Discussion

are thus interesting for security-related static analysis. However, SUIF2 is not under active development.

# 7. Closing Remarks

This thesis has evaluated whether intermediate representations of a real-world compiler can be reused for security-related static analysis. A method to export commonly used data structures has been developed. The solution has been evaluated using a well-known program analysis algorithm. This demonstrates that the exported data structures accelerate the implementation of such algorithms significantly. The intermediate representations of three software packages have been exported to evaluate the applicability to real-world software. Although the resulting data structures are large, their parsing time is remarkably low.

## 7.1. Future Work

The current version of rxd has two drawbacks. Object-oriented languages are represented in a verbose imperative way. In Section 7.1.1, possible solutions to this problem are outlined.

Rxd is an external extension to the GCC, so that it has to be adopted to changes in GIMPLE or other internals of the compiler. A future version of GCC possibly exports intermediate representations to perform link-time optimizations. Impacts of this development to the presented work are discussed in Section 7.1.2.

### 7.1.1. Object-oriented Languages

The current representation of object-oriented languages in rxd is not satisfying. Object-oriented features are translated into sequences of imperative statements, as discussed in Section 6.3.3. This approach is necessary for a compiler, because it has to generate executable code. In contrast, the goal of a static checker is to analyze the possible behaviour of a program. This can be exemplified using the representation of method calls. The run-time implementation of a method call, i.e. the lookup of the correct function call in the vTable, is not interesting for high-level analyses. A set of possible target methods is much more desirable for this class of analyses.

A possible approach to this problem is to recreate higher-level information from the output of rxd. A tool could analyze the vTable lookup and determine which functions may possibly be called. This information could be used to replace the imperative implementation of method calls inside the XML files, thus making the information available for further analyses. The detection and replacement of method calls gets difficult when exceptions may occur during the look-up of a method, because one has to deal with several possible execution paths.

## 7. Closing Remarks

An alternative is to use an object-oriented parser, e.g. the C++ front end of GCC. Object-oriented features are available, but no additional data structures, such as control flow graphs are generated. The regular structure of GIMPLE is helpful for both generating additional data structures and analyses in general. An object-oriented GIMPLE, i.e. the regular structure of GIMPLE combined with object-oriented tree codes might be an interesting approach when an object-oriented parser is used.

Another problem when analyzing C++ programs is the usage of templates. Templates generate source code during the compilation, e.g. to generate type-safe lists of objects. In `cp-tree.h`<sup>1</sup>, it is noted that the nowadays heavy usage of templates creates problems with the internal representation of classes inside the C++ front end. The assumption that there are possibly only few classes inside a C++ file is no longer true, when templates are used. Thus the GIMPLE representation of C++ programs explodes when templates are used, as it can be seen for Heatload in Section 6.2.3. The Pivot framework as outlined in [SR05] is a high-level static analysis framework for C++, supporting advanced features, such as templates. An evaluation of this framework for security-related static analysis would be interesting, but Pivot is not available yet.

### 7.1.2. Link-time Optimization

Recently, the link-time optimization project [MZ05] was started within the GCC to enable optimizations at link-time, i.e. intermodule optimizations. Therefore, an intermediate representation that contains all information necessary to finalize a compilation has to be exported to the hard disk. At the moment, it is not decided yet which intermediate representation will be used. One of the candidates is GIMPLE as a control flow graph, perhaps in SSA form.

When GIMPLE is chosen, it would be interesting to evaluate how the format of these intermediate files could be used for security-related static analysis. Using an officially-supported format is appealing. First, it is included in every version of the GCC, so it is not necessary to install and use a patched version the compiler. Additionally, future changes to the internal formats of GCC will be supported by this format.

## 7.2. Conclusion

The usage of compiler intermediate representations for security-related static analysis is promising. Data structures available are either directly useful to implement analyses, or helpful to generate additional data structures. The extension presented in this work provides easy access to these intermediate representations. This accelerates the implementation of new analysis approaches, as the proof of concept analysis demonstrated. An additional advantage is the regular structure of GIMPLE, which enables simpler implementations of analysis algorithms.

The XML format allows the implementation of new approaches in all high-level languages where an XML parser is available. Accessing elements inside the XML files is

---

<sup>1</sup>This file defines the data structures used to represent C++ parse trees.

comfortable using XPath, as described in Section 6.1.2. The validation has shown that parsing of large XML files is fast enough to be usable for real-world programs. The on-disk format is also useful to combine several analyses. I.e. the result from one analysis could be stored inside the XML files and thus reused by other analyses.

Compilers and security-related static analysis tools have different goals regarding object-oriented languages. Compilers have to create executable code, thus object-oriented features are translated into imperative statements that are easy to translate to machine code. This complicates the analysis of object-oriented programs, although the information could be recovered from the GIMPLE representation, as proposed in Section 7.1.1.

## 7. *Closing Remarks*

# A. Integer Range Analysis Algorithm

```
1 procedure abstract interpretation (graph);
2 begin
3   for each arc of graph do local context (arc) :=  $\Phi$  repeat;
4   execution paths := exit-arc (entry-node (graph)), junctions =  $\emptyset$ ;
5   while (execution paths !=  $\emptyset$ ) do
6     while (execution paths !=  $\emptyset$ ) do
7       input arc := choose (execution paths);
8       execution paths = execution paths - input arc,
9       node = final-end (input arc);
10      case node of
11        assignment node  $\rightarrow$ 
12          assign output context (exit-arc(node),
13            Interpret(node, local context(input arc)));
14        test node  $\rightarrow$ 
15          (Ct, Cf) := Interpret(node, local context(input arc));
16          assign output context (true-exit-arc (node, Ct);
17          assign output context (false-exit-arc (node, Cf);
18        simple or loop junction node  $\rightarrow$ 
19          junctions:= junctions  $\cup$  node;
20        exit node  $\rightarrow$  ;
21      end;
22    repeat;
23  for each junction node of junctions do
24    output context =  $\bar{\cup}$  local context (input arc);
25    input arc  $\in$  entry-arcs(junction node)
26    if  $\neg$  (output context  $\leq$  local context (exit-arc(junction node)))
27      then
28        case junction node of
29          simple junction node  $\rightarrow$ 
30            assign output context (exit-arc(junction node), output context);
31          loop junction node  $\rightarrow$ 
32            assign output context (exit-arc(junction node),
33              local context(exit arc(junction node)  $\bar{\nabla}$  output context) ;
34          end;
35        fi;
36      repeat;
37    junctions =  $\emptyset$ 
38  repeat;
39 return;
```

## A. Integer Range Analysis Algorithm

```
40
41 procedure assign output context(output arc, output context);
42   if  $\neg$  (output context  $\leq$  local context (output arc)) then
43     local context (output arc) := output context;
44     execution paths := execution paths  $\cup$  output arc;
45   fi;
46 end;
```

## B. CD-ROM

All source code used in this thesis is available on a CD-ROM that can be found at the end of this thesis. The content can be summarized as follows:

**abstract\_interpretation** The source code of the proof of concept analysis, as well as the files used in the example can be found here.

**evaluation** The shell scripts used during the practical evaluation and the generated log files can be found in this directory.

**rxd** This directory contains the original version of GCC, a patched version of GCC which includes rxd, and the rxd patch itself.

**xmlSchema** The XML Schema files can be found in this directory, together with a spread sheet summarizing the current implementation status for each GIMPLE tree-code.



# Bibliography

- [ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison Wesley, 1988.
- [Bri06] British Computer Society Specialist Interest Group in Software Testing. Glossary of terms used in software testing. [http://www.testingstandards.co.uk/bs\\_7925-1\\_online.htm](http://www.testingstandards.co.uk/bs_7925-1_online.htm), June 2006. Working Draft Version 6.3.
- [CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
- [CC77a] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [CC77b] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.
- [CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 159–178, London, UK, 2002. Springer-Verlag.
- [CCF<sup>+</sup>05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP'05*, 2005.
- [CDW04] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185, San Diego, CA, February 4–6, 2004.
- [CFR<sup>+</sup>91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form

## Bibliography

- and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
- [CW02] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA, 2002. ACM Press.
- [Dev04] IBM XL Compiler Development. Code optimization with the IBM XL compilers. Technical report, IBM, 2004.
- [DuP06] James Michael DuPont. Introspector project. <http://introspector.sourceforge.net/>, August 2006.
- [Ebn06] Dietmar Ebner. XMLDUMP - GCC tree dumps. <http://www.complang.tuwien.ac.at/cd/ebner/>, August 2006.
- [ECCH00] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, 2000.
- [EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.
- [Eng06] Dawson Engler. Weird things that surprise academics trying to commercialize a static checking tool, August 2006. Invited Talk at SPIN05 and CONCUR05.
- [FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
- [Fre06a] Free Software Foundation Inc. GCC home page. [gcc.gnu.org](http://gcc.gnu.org), June 2006.
- [Fre06b] Free Software Foundation Inc. GNU C library. <http://www.gnu.org/software/libc/>, June 2006.
- [GaSJS05] R. Grosu, X. Huang and S. Jain, and S.A. Smolka. Open source model checking. In *Proc. of SoftMC'05*, 2005.
- [GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 108–124, New York, NY, USA, 1997. ACM Press.

- [HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, New York, NY, USA, 2002. ACM Press.
- [HDE<sup>+</sup>92] Laurie J. Hendren, C. Donawa, Maryam Emami, Guang R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *LCPC*, pages 406–420, 1992.
- [HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM Press.
- [Kit06] Kitware, Inc. GCC-XML. <http://www.gccxml.org>, August 2006.
- [Lac06] Patrice Lacroix. RTLcheck. <http://rtlcheck.sourceforge.net/>, August 2006.
- [Lam06] Monica Lam. An overview of the SUIF2 system. <http://suif.stanford.edu/>, August 2006.
- [Lan92] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.
- [Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [Mue06] Jan Moritz Muehlenhoff. Static source code vulnerability detection using intermediate representations of gcc. Master’s thesis, Universität Bremen, 2006.
- [MZ05] Mark Mitchell and Kenny Zadeck. Link-time optimization in gcc: Requirements and high-level design. <http://gcc.gnu.org/projects/lto/lto.pdf>, November 2005.
- [NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.
- [NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
- [Nov03] Diego Novillo. TreeSSA a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers’ Summit*, pages 181–193, 2003.

## Bibliography

- [Nov06a] Diego Novillo. Gcc - an architectural overview, current status and future directions. In *Proceedings of the Linux Symposium*, volume Volume Two, 2006.
- [Nov06b] Diego Novillo. Memory ssa - a unified approach for sparsely representing memory operations. PRELIMINARY DRAFT <http://people.redhat.com/dnovillo/pub/mem-ssa.pdf>, February 2006.
- [Poe06] Lennart Poettering. Heatload home page. <http://0pointer.de/lennart/projects/heatload/>, July 2006.
- [Sec06] Secure Software. Rats - rough auditing tool for security. [http://www.securesoftware.com/resources/download\\_rats.html](http://www.securesoftware.com/resources/download_rats.html), June 2006.
- [Sen06] Sendmail Consortium. Sendmail home page. <http://www.sendmail.org/>, July 2006.
- [SR05] Bjarne Stroustrup and Gabriel Dos Reis. Supporting sell for high-performance computing. In *Workshop on Languages and Compilers for Parallel Computing*, 2005.
- [StGDC05] Richard Stallman and the GCC Developer Community. Gnu compiler collection internals (for gcc version 4.1.0). Technical report, Free Software Foundation, 2005.
- [The06] The Eclipse Foundation. Eclipse java development tools. <http://www.eclipse.org/jdt/>, August 2006.
- [VBKM00] John Viega, J. T. Bloch, Y. Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *ACSAC*, 2000.
- [W3C06a] W3C. Document object model (DOM). <http://www.w3.org/DOM/>, August 2006.
- [W3C06b] W3C. Extensible markup language (XML). <http://www.w3.org/XML/>, August 2006.
- [W3C06c] W3C. Xml path language (XPath). <http://www.w3.org/TR/xpath>, August 2006.
- [W3C06d] W3C. XML Schema. <http://www.w3.org/XML/Schema>, August 2006.
- [Wad90] Vance E. Waddle. Production trees: a compact representation of parsed programs. *ACM Trans. Program. Lang. Syst.*, 12(1):61–83, 1990.

- [WF05] John Wilander and Pia Fak. Pattern matching security properties of code using dependence graphs. In *Proceedings of the 1st International Workshop on Code Based Software Security Assessments (CoBaSSA 2005)*, Pittsburgh, Pennsylvania, November 2005.
- [WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.
- [Whe06] David A. Wheeler. Flawfinder home page. <http://www.dwheeler.com/flawfinder/>, June 2006.
- [Wil05] John Wilander. Modeling and visualizing security properties of code using dependency graphs. In *Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden*, 2005.
- [XNHA05] Y. Xie, M. Naik, B. Hackett, and A. Aiken. Soundness and its role in bug detection systems (position paper). In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

## *Bibliography*

## **Erklärung**

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

---

Thilo Mende