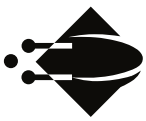
Effective Bug Finding

Iago Abal Rivas

Advisors: Andrzej Wąsowski and Claus Brabrand
Submitted: March 1st, 2017



IT University
of Copenhagen

Abstract

Lightweight bug finders (also known as *code scanners*) are becoming popular, they scale well and can find simple yet common programming errors. It is now considered a good practice to integrate these tools as part of your development process. The Linux project, for instance, has an automated testing service, known as the *Kbuild robot*, that runs a few of these code scanners.

In this project, I have carefully studied tens of historical Linux bugs, and I have found that many of these bugs, despite being conceptually simple, were not caught by any code scanning tool. The reason is that, by design, code scanners will find mostly superficial errors. Thus, when bugs span multiple functions, even if simple, they become undetectable by most code scanners. The studied set of historical bugs contained many of such cases.

This PhD thesis proposes a bug-finding technique that is both lightweight and capable of finding deep interprocedural *resource manipulation bugs*. The core of this technique is a *shape-and-effect* analysis for C, that enables efficient and scalable inter-procedural reasoning about resource manipulation. This analysis is used to build an abstraction of the program. Then, bugs are found by matching temporal bug-patterns against the control-flow graph of this program abstraction.

I have implemented a proof-of-concept bug finder based on this technique, EBA, and confirmed that it is both *scalable* and *effective* at finding bugs. On a benchmark of historical Linux double-lock bugs, EBA was able to detect significantly more bugs, and more complex, than two other baseline tools. EBA was able to analyze nine thousand files of device drivers from Linux-4.7 in less than half an hour, in which time it uncovered five previously unknown bugs. So far, EBA has found *more than a dozen double-lock bugs* in Linux 4.7–4.10 releases, most of them already confirmed and many fixed.

Resumé

Automatiske letvægts-fejlfindere (også kaldet kode-scannere) er ved at blive populære. De er skalérbare og kan finde simple, men ofte forekommende programmeringsfejl. Det er efterhånden betragtet som god praksis at integrere disse fejlfindingsværktøjer i udviklingsprocessen. Linux-projektet har eksempelvis tilknyttet en automatisk test-service kaldet "Kbuild robot", som afvikler en række kode-scannere.

I dette projekt har jeg nærstuderet et halvt hundrede historiske fejl i Linux og fundet frem til at mange af disse - selv om der er tale om begrebsmæssigt simple fejl - ikke kan findes med nuværende kode-scannere. Årsagen til dette er at kode-scannere - som konsekvenser af designvalg - går efter de mest overfladiske fejl. Når der er tale om fejl der involverer flere funktioner, så kan de ikke findes af kode-scannere, også selv om der er tale om ganske simple fejl - faktisk indeholdte den mængde af historiske fejl vi studerede adskillige sådanne fejl.

Denne Ph.D.-afhandling foreslår en ny fejlfindingsteknik, som er både letvægts samt i stand til at finde dybe interprocedurelle *ressoucemanipulationsfejl*. Denne fejlfindingsteknik er baseret på *form og effekt* analyse for C, som muliggør skalérbar interprocedurel ræsonnemang omkring manipulation af ressourcer. Analysen bruges til at bygge en abstraktion af programmet. Fejl findes efterfølgende ved at anvende temporal mønstergenkendelse på programabstraktionens graf over kontrol-flow.

Jeg har implementeret en prototype-fejlfinder ved navn EBA, baseret på denne teknik og bekræftet at denne teknik er både skalérbar samt effektiv til at finde fejl. På en samling af historiske dobbeltlåsfejl i Linux var EBA i stand til at finde signifikant flere - og mere komplekse - fejl sammenlignet med to andre fejlfindingsværktøjer. EBA var i stand til at analysere ni tusinde device-driver-filer fra Linux-4.7 på mindre end en halv time samt finde fem hidtidigt ukendte reelle fejl. EBA har indtil nu fundet mere end et dusin dobbeltlåsfejl i i Linux 4.7–4.10 - de fleste af dem er allerede bekræftet samt udbedret.

† This abstract in Danish has been automatically generated, and it is intended for the (possibly empty) set of people fulfilling the following requirement: knows Computer Science, yet is unable to read English.

If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.

Edsger W. Dijkstra

Acknowledgements

Apart from our efforts, the success of any project depends largely on the encouragement and guidelines of many others.

I would like to thank the Danish Council for Independent Research, for funding this work under the Sapere Aude 2 / VARIETE grant.

I would like to thank my advisors Andrzej Wąsowski and Claus Brabrand for their help, support, and guidance over the past three years and a half. We have made a great team.

I would like to thank to all those who contributed to my education. I owe a special debt of gratitude to Jorge Sousa Pinto, a mentor and a friend, who introduced me to static program analysis.

Last but not least, I would like to thank my parents, my wife, and my son, for their unconditional love and support. I love you.

Contents

Chapter 1

Introduction

1.1 Context

This PhD project is part of a wider research project, VARIETE (Variability in Portfolios of Dependable Systems), funded by the Danish Council for Independent Research (DFF). The purpose of the VARIETE project has been to develop techniques to approach the verification of highly-*configurable* dependable *software* systems.

Configurable software (also known as, *variable software*) arises as a necessity to adapt software systems to different demands, like hardware platforms or usage scenarios. Software variability supports the development of *program families*. A program family is a piece of parameterized software, from which many software systems (*program variants*) can be derived by setting configuration options (*features*). Software variability is a cost-effective strategy to develop and maintain multiple variations of a core software system, that if conceived individually would multiply development and maintenance costs [PBL05].

Variability is ubiquitous in system-level software (e.g. operating and embedded systems), and in any scenario where performance and low resource utilization are required. Such software is mostly written in C, and implements compile-time variability through the C preprocessor (CPP). Features are represented by macro symbols —often identified by a prefix like `CONFIG_`, and variable code is surrounded by CPP `#if` conditional directives over those symbols. The code associated with a disabled feature is removed by CPP during compilation, thus reducing the final binary size. The Linux kernel, with thousands of configuration options, is a prime example of a highly-configurable software system [LSB+10].

```
1  void foo(void) {
2          int *p = malloc(sizeof(int)); // allocation
3          if (!p) return;
4
5          printf("%d",p);
6
7          #ifdef CONFIG_FREE           // disabled
8          free(p);                     // not compiled
9          #endif
10
11         return;                      // BUG
12 }
```

Figure 1.1: Example of a program family with a variability bug. A memory-leak error occurs upon the return of the function whenever FREE is disabled.

Figure 1.1 should serve to illustrate the kind of configurable software that we are interested in. Function `foo` allocates memory for an integer variable `*p` in line 2, and prints the memory address of that variable in line 5. Since the memory assigned to `p` will not be accessible after `foo` returns, returning without freeing `p` constitutes a memory leak. This is the case if feature FREE is disabled, since in such configuration the `free` statement in line 8 will not be compiled. (If FREE is enabled, then `p` is freed and there is no memory leak.) A bug like this one, which is present in some configurations but not in others, is a *variability bug* [ABW14].

Whereas variable software is widespread, we lack tools capable of verifying all derivable variants of a program family. It is completely infeasible to verify each configuration individually: highly-configurable systems are parameterized by tens of configuration options, and involve billions of different variants. Even if we could verify one program variant per second, it will still take us 30 years to verify a billion variants. Some have proposed the use of *family-based analyses* [TAK+14], that tackle this problem by considering all configurable program variants as a single unit of analysis, instead of analyzing the individual variants separately [KA08, ASW+11, CHSL11, BTR+13, BRT+13].

For the time being, the verification of configurable software is approached through *configuration sampling*. A sample of the many possible configurations is selected, according to some criterion, and each program variant is verified in isolation. For instance, the *Kernel Instant bug*

testing Service (KIS) uses a server farm to test 141 configurations of the Linux kernel per day [CWY+13]. Yet, due to the discrete nature of software, the successful verification of a program variant cannot be used to predict the correctness of even a slightly different variant. Hence, if the right configuration is not hit by sampling, a critical bug can be missed.

The development of improved verification techniques and methods for configurable software is one of the main goals of VARIETE. A small team of people, myself included, have worked together in pursuing this goal. First of all, we have studied the nature of feature-interaction bugs in configurable software [ABW14, MFBW16, MBW16]. Then, we have worked on two different approaches to verifying configurable software. One research direction has focused on family-based analyses [MDBW15, DBW15]. The other research direction has focused on exploiting single-program static analyzers to verify configurable software [PS08].

My mission has been to investigate effective ways of finding bugs in large-scale systems-level software [ABW17], a problem that is elaborated in the following section 1.2. Other colleagues have worked on making a single-program verification technique —such as the one developed in this thesis— work, efficiently, on a family of C programs by *rewriting variability* [IMD+17].

1.2 Problem

Bugs are easier and cheaper to fix when they are found early. Hence, we would like our verification tools to run directly on developers' computers, be part of their development environments, and provide them with continuous feedback on their code. However, the reality today is that many verification tools are not fast enough for this purpose, taking hours to analyze just a few tens of thousands lines of code. At best, they can be run once a day, as part of a nightly build. At worst, in some scenarios, they are too slow to be run at all.

Unsurprisingly, many developers rely solely on their compilers to check their code for errors. (Testing is also common, but some kinds of software, such as device drivers, can be difficult to test in an automated fashion.) Some developers do use lighter-weight static bug-finding tools, so-called *linters* (i.e., Lint-like tools [Joh78]) or *code scanners*. [1] Linters run

[1] To my knowledge, there is no standard classification of static analysis tools.

fast and can fit into developers' work-flow, but mostly find relatively simple and shallow bugs.

Due to timing and resource constraints, linters are the only practical verification solution for some projects. This problem is amplified when software variability is taken into consideration. For instance, the aforementioned KIS instant bug testing service for Linux relies exclusively on linters. KIS could not, otherwise, statically analyze all of the commits pushed daily to the many Linux repositories, under 141 different configurations, within a day.

In the VARIETE project, our objective has been to make static verification of highly-configurable software systems, like the Linux kernel, widely affordable. We are consequently interested in lightweight Lint-like techniques, but we would like to count with tools that uncover more complex bugs that linters currently do. (After all, the VARIETE project concerns the verification of dependable software.) Investigating the balance between the scalability of tools, and the qualities of the bugs found by these tools, has been a priority for us, and the goal of this PhD thesis.

In summary, my problem has been to answer the following research question:

> *How can we effectively analyze large-scale systems-level software for non-trivial bugs and do it as fast as linters?*

1.3 Method

There is no better way to address scalability, than to consider it from the very beginning. For that reason, I have selected and concentrated my efforts on a single yet very large subject, the Linux kernel. Linux is the ideal subject of study for several reasons: it is open source, comparatively well documented, and it is a very large and complex piece of software. With more than ten thousand configuration options, Linux is also a prime example of a highly-configurable software system [LSB+10].

In order to develop a solution for the problem previously discussed, I followed a simple problem-oriented research method with three steps: [2]

Step 1: Study of a collection of historical Linux bugs. First of all, I collected a sample of 43 historical Linux bugs, and analyzed each bug both from a programming-language and variability perspective (cf. Chapter 3). The goal was two-fold. For the VARIETE project as a whole, we wanted to understand the nature of bugs in highly-configurable systems-level software. Before this study, such understanding did not properly exist. For my PhD problem in particular (cf. Sect. 1.2), I wanted to identify common bug patterns, and understand the limitations of current bug finding techniques for recognizing such bugs.

Step 2: Development of the bug-finding technique. Previously, in Step 1, I identified a class of non-trivial bugs that were not caught by state-of-the-art code scanners. Subsequently, I devised an strategy to find these bugs statically, yet efficiently (cf. Sect. 3.8). The key ingredients for efficient bug finding are *abstraction* and *decomposition*. The challenge was to find a program abstraction that was cheap to compute, but provided enough information to analyze programs in a modular fashion (cf. Chapter 4). I formally developed this strategy into a lightweight bug-finding technique (cf. Chapter 6).

Step 3: Evaluation of the bug-finding technique. I implemented a prototype, the *Effect-Based Analyzer* (EBA), to prove that my bug-finding technique was realizable. During early stages of development, I used the bugs collected in Step 1 as a benchmark, to test and guide the design of EBA. I used this prototype as a proxy to evaluate the scalability and effectiveness of the proposed technique. The evaluation consisted in comparing the performance of EBA against similar baseline tools, on the task of analyzing Linux device drivers for bugs.

1.4 Thesis

The study of 43 historical bugs in Linux revealed that, despite the many resources dedicated to quality assurance, the Linux kernel continues to

[2]Note that this PhD thesis is only concerned with the analysis of individual program variants, see Sect. 1.1.

suffer from conceptually simple bugs. A significant amount of these bugs are related to the mis-manipulation of resources, such as accessing a de-allocated memory region, or double-acquiring a non-reentrant lock. Code scanners have been used extensively in an attempt to remove these bugs from the Linux kernel [LMP09, CWY+13].

In particular, Linux-tailored code scanners Sparse, Coccinelle, and Smatch, are well-known within the Linux kernel community. These three tools run on the Linux source tree "out the box", and are fast and also reasonably effective at finding certain classes of bugs. For instance, Linux commits ca9fe15 [3] and 65582a7 fix locking bugs found by two of these tools. But code scanners are largely limited to syntactic intra-procedural analysis, and they mostly find shallow bugs, not involving nested function calls.

Remarkably, my study of historical Linux bugs shows that:

> *Bugs often cross the boundaries of a single function, and thus are out of the reach of most code scanners.*

For instance, commits: 1c17e4d (read of uninitialized data), 6252547 (null pointer dereference), 218ad12 (memory leak), and d7e9711 (double lock), are all simple yet inter-procedural resource manipulation bugs. This observation has been key in developing this PhD work.

Figure 1.2 shows a simplified version of one of these historical bugs in Linux, the double-lock bug fixed by commit d7e9711760a. Function `add_dquot_ref` causes a potential deadlock by double-acquiring a non-reentrant *spin lock*. The first lock acquisition occurs in line 10, and the second occurs in line 4 after calling function `inode_get_rsv_space` (in line 15). For the error to occur, both conditionals (lines 3 and 11) must evaluate to *false*—i.e., take the *else* branch.

Traditionally, we have relied on heavyweight static analyzers based on abstract interpretation, inter-procedural data-flow analysis, or symbolic execution, to intercept deep inter-procedural bugs. These analyses are often not compositional, or rely on computationally expensive abstractions; and hence present scalability challenges, having seen little adoption in practice.

[3] See https://github.com/torvalds/linux/commit/hash with hash replaced by the identifier.

```
1  void inode_get_rsv_space(struct inode *inode)
2  {
3      if (*) return;
4      spin_lock(&inode->i_lock); // 2nd lock
5      spin_unlock(&inode->i_lock);
6  }
7
8  void add_dquot_ref(struct inode *inode)
9  {
10     spin_lock(&inode->i_lock); // 1st lock
11     if (*) {
12         spin_unlock(&inode->i_lock);
13         return;
14     }
15     inode_get_rsv_space(inode); // call
16     spin_unlock(&inode->i_lock);
17 }
```

Figure 1.2: A simplified version of a double-lock bug in Linux fixed by commit d7e9711760a.

I argue that a wide class of resource manipulation bugs can be efficiently uncovered with simple Lint-like techniques, even when the manipulation of resources spans multiple functions. The recipe consists in performing modular program analysis, supported by lightweight abstractions. I propose that such program abstractions are inferred by a flow-insensitive type-and-effect analysis [Luc87, NN99].

This PhD thesis shows that:

> *Flow-insensitive side-effect analysis can be used to construct lightweight program abstractions, that enable efficient interprocedural reasoning about the manipulation of resources. A software model-checker can use the inferred side-effect summaries to prune the control-flow graph, and effectively analyze very large code bases, such as the* Linux *kernel.*

To prove this, I have developed a bug-finding technique that consists in matching temporal bug patterns against shape-and-effect program

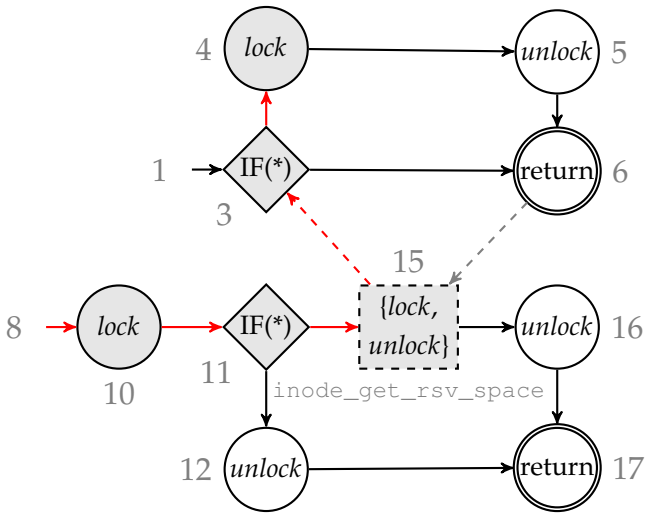


Figure 1.3: An illustration of our bug-finding technique for the double-lock bug in Figure 1.2. The figure shows the associated CFG annotated with lock and unlock effects. The numbers next to the CFG nodes show corresponding line numbers. The gray nodes visualize the (red) path, via the function call in line 15, to the double-lock (in line 4).

abstractions. The core of this technique, is a shape-and-effect analysis based on the work of Talpin and Jouvelot on polymorphic type-and-effect inference [TJ92]. This analysis infers types that approximate the *shape* of data in memory—hence the term *shape*-and-effect analysis, and also computational *effects* that describe how data is manipulated by the program. The inference algorithm is a small variation of the classic Damas-Milner's *Algorithm* $\mathcal{W}$ [DM82].

Next, the inferred shape-and-effect information is superimposed on the control-flow graph (CFG), what results in the so-called *shape-and-effect abstraction* of the program. In this abstraction, each program expression and statement is described by a set of computational effects. (The set of inferred effects is extensible and depends on the *bug checker*, for instance, to find the bug of Fig. 1.2 we are interested in the acquisition and release of locks.) Bugs are found by matching temporal bug patterns against this abstraction, using a standard model-checking algorithm. Function calls that, by their effect signature, are deemed irrelevant for the analysis, are treated as opaque expressions. Function calls that manipulate a resource of interest (e.g. a lock), may be inlined on-demand.

Figure 1.3 shows a simplification of the effect-decorated CFG associated with the program of Fig. 1.2. For the analysis of double-lock bugs, the effect abstraction is concerned with the locking effects on `&inode->i_lock`. The gray nodes, and edges between them, mark an execution path leading to the double lock. The call to `inode_get_rsv_space` is abstracted by a flow-insensitive summary of effects (the set $\{lock, unlock\}$). In line 15, from the effect signature of function `inode_get_rsv_space` alone, it is unclear as to whether the acquisition of `&inode->i_lock` happens *before* or *after* its release. In such an inconclusive scenario, the model-checker proceeds by taking the function call (represented as following the gray dashed lines), which unveils the double-lock bug in line 4.

In summary, my contributions are:

- *Identification and in-depth analysis of 43 bugs in the* Linux *kernel* (Chapter 3). These bugs comprise common types of errors in systems-level software, and cover different types of feature interactions. This study has led to the creation of *The Variability Bug Database* (http://VBDb.itu.dk/), which encompasses a detailed data record about each bug.

- *A bug-finding technique that is both effective and efficient* at finding a wide class of resource manipulation bugs, even when these involve deep function call chains (Chapter 6). This technique relies on *an adaptation of Talpin-Jouvelot's [TJ92] polymorphic type-and-effect inference system to the* C *language* (chapters 4 and 5), that is used to infer shape-and-effect abstractions of C programs.

- *A proof-of-concept prototype of the shape-and-effect system, and the proposed bug-finding technique*: *The Effect-Based Analyzer*, EBA (http://www.iagoabal.eu/eba/). EBA can often analyze individual Linux files for bugs in a few tenths of a second, and all the x86 *allyesconfig* Linux kernel drivers in about half an hour (Chapter 7). So far EBA has uncovered a dozen of previously unknown double-lock bugs in Linux-4.7.

Outline. I proceed discussing past and present work on program static analysis and bug finding (Chapter 2). Next, I present a qualitative

study of 43 historical Linux bugs (Chapter 3), both from a programming-language and (in the context of VARIETE) variability perspective. This study motivates the core of this PhD thesis: an effective and efficient bug-finding technique. This technique is based on *shape-and-effect* abstractions (Chapter 4). I describe how to infer these abstractions (Chapter 5), how to use them to find interprocedural resource mismanipulation bugs (Chapter 6), and implement EBA: a proof-of-concept implementation of the proposed bug-finding technique. Finally, I evaluate EBA (Chapter 7), provide directions for future work, and draw my final conclusions (Chapter 8).

Chapter 2

Related Work

First, I discuss previous studies of software bugs (Sect. 2.1), and why the study of 43 historical Linux bugs of Chapter 3 was needed. Second, I describe previous work on side-effect inference and pointer analysis (Sect. 2.2), and motivate the choice of the work of Talpin and Jouvelot [TJ92] as the basis of the shape-and-effect system of Chapter 4. Last but not least, I briefly survey static checking of resource manipulation (Sect. 2.3), discuss the pros and cons of each major family of tools, and related those to the trade-offs made in the design of the bug-finding technique of Chapter 6.

2.1 Empirical study of software bugs

This section discusses work related to the study of 43 historical bugs in Linux, and the construction of the VBDb database, described in Chapter 3.

ClabureDB is a database of bug reports for the Linux kernel with similar purpose to that of VBDb [SST13], albeit ignoring variability. The main strength of ClabureDB is its size—the database is automatically populated using existing bug finders, as of February 2017, it contains 221 confirmed Linux bugs, and 850 false positives. VBDb is comparatively small, with only 43 Linux bugs documented. [1] We had to populate it manually, given that we are interested in the study of bugs that state-of-the-art tools cannot find efficiently. Incidentally, since no suitable bug finders

[1] Jean Melo, Stefan Stanciulescu, and Márcio Ribeiro have contributed 55 more bugs to VBDb from three other open-source projects: the BusyBox *NIX utilities, the Marlin 3D-printing firmware, and the Apache webserver [AMS+17].

handling variability exist, none of the bugs in VBDb is adequately covered in ClabureDB. The main strength of VBDb is in the detailed analysis of each bug, and the derived artifacts. We have provided a record with information enabling non-experts to rapidly understand the bugs and benchmark their analyses. This includes a simplified version of each bug where irrelevant details are abstracted away, along with explanations and references intended for researchers with limited knowledge of the Linux kernel.

Palix et al. [PTS+11] reproduced an empirical study measuring the frequency of certain kinds of bugs in the Linux kernel as of 2001, ten years later, in 2011, to reevaluate and investigate the evolution of bugs in Linux over a decade. The results are available in a public archive.[2] This study has identified a series of Linux-specific bugs and rules, such as *"do not use floating point in the Linux kernel"*. They searched for pre-defined types of bugs in the Linux kernel source code, encoding these rules as Coccinelle scripts. The bugs found were not confirmed by Linux developers, and it is unknown how many of them were, in fact, real bugs. In turn, my focus has been on qualitatively understanding the complexity and nature of (variability) bugs, for which I have mined and subsequently analyzed 43 historical (confirmed and fixed) bugs in Linux.

Nadi et al. mined the Linux repository to study the occurrence and nature of *variability anomalies* in Linux [NDT+13]. An *anomaly* is a *mapping* error, such as mapping code to an invalid configuration, or code that is mapped to nonexistent features. They could analyze a large number of commits automatically by using the Undertaker tool [TLSSP11], which is capable of finding such variability anomalies. While I conduct my study in a similar fashion, I have focused on semantic errors in the code—and I had to analyze commits manually.

Apel et al. use a model-checker to find feature interactions in a simple email client [ASW+11]. They used a technique known as *variability encoding* (or *configuration lifting* [PS08]). This consists in encoding features as Boolean variables, and transforming CPP conditional directives (`#if`, etc.) into `if` conditional statements. I focused on understanding the nature of variability bugs widely. This cannot be done with a model-checker searching for a particular class of feature interactions. (Tools are fundamentally biased towards finding specific kinds of bugs.) Understanding variability bugs should lead to building scalable bug finders, enabling studies like [ASW+11] to be run for Linux in the future.

[2] `http://faultlinux.lip6.fr/`

Medeiros et al. studied *syntactic* variability errors [MRG13]. They used a variability-aware C parser [KKHL10] to automate their bug finding, and exhaustively found *all* syntax errors. They found only a few tens of errors in 41 program families, suggesting that syntactic errors caused by variability are rare in practice. These 41 subjects are quite diverse, including: web-servers, such as Apache; version-control systems, such as CVS; text editors, such as vim; programming-language interpreters, such as Lua; etc. These are small to medium-size pieces of software, which sum up about 4 MLOC and nine thousand files—Linux `drivers/` alone has about 8 MLOC and 13 thousand files. Yet, the main difference with my work is that I focus on the wider category of more complex *semantic* errors.

Tian et al. studied the problem of distinguishing bug fixing commits in the Linux repository [TLL12]. They use semi-supervised learning to classify commits according to tokens in the commit log and code metrics extracted from the patch contents. They significantly improve recall (without lowering precision) over keyword-based methods like mine. In my study of Linux bugs, most of the time was invested in analyzing commits, not in finding commits to analyze in the first place, so I found a simple keyword-based method sufficient.

Yin et al. collected hundreds of errors caused by mis-configurations in open source and commercial software [YMZ+11]. They consider systems in which parameters are read from configuration files, as opposed to systems configured statically. The main difference is that they document errors from the *user* perspective, as opposed to (in my case) *programmer* perspective.

Padioleau et al. studied collateral evolution in Linux device drivers, following a method close to mine [PLM06]. Collateral evolution occurs when existing code is adapted to API changes. They developed a tool to heuristically identify collateral evolutions by analyzing bug-fixing commits, and then manually selected 72 cases for a more careful analysis. Before this study, the authors already had a good understanding of the nature of collateral evolutions. Thus, in [PLM06] their approach is *quantitative*, they classify and estimate the amount of collateral evolutions happened in between Linux 2.2 and 2.6 releases. Note that this study concerns a very specific kind of bugs, when drivers code is broken due to changes to kernel interfaces. My intent was to qualitative study (variability) bugs widely, to better understand the nature of these bugs. Such an understanding did not previously exist.

From the VARIETE perspective, the study of 43 variability bugs in Linux has increased our understanding about how feature-interactions lead to (sometimes very subtle) software bugs, in highly-configurable systems-level software. Before this work, most feature-interaction bugs have been identified, documented, and published in the telecommunication domain [CKMRM03]. VBDb provides a unique collection of variability bugs that, I believe, will be useful in designing future variability-aware tooling (e.g., [IMD+17]). These bugs are good examples of the many C intricacies and idioms that make the design of static analyzers challenging (and fun!). Crucially, it was the careful study of these bugs that made me recognize the so-called *resource mis-manipulation* category of bugs, ultimately leading to this thesis.

2.2 Side-effect analysis

This section discusses work related to the shape-and-effect inference algorithm described in Chapter 6. I begin by discussing type-and-effect systems, then pointer analysis.

2.2.1 Types and effects

The notion of type-and-effects was introduced by John M. Lucassen in his Ph.D. thesis *"Types and Effects: Towards the Integration of Functional and Imperative Programming"* (1987) [Luc87]. Lucassen proposed that a type-and-effect system could replace the side-effect flow-analysis performed by optimizing compilers [Ban78, Ban79, CK84, CK88]. Side-effect analysis is used to compute which memory locations are accessed or updated by a function call. Traditionally, this information is used by compilers to determine whether it is legal to perform certain code optimizations. He presented a polymorphic effect system that could perform inter-procedural side-effect analysis, in the presence of first-class functions, or function pointers.

His work was the origin of FX [JG88] by Jouvelot and Gifford, a language designed for automatic parallelization, where effects are used to determine the non-interference of expressions. Two expressions do not interfere, if they read from and write to disjoint sets of memory locations. Consequently, non-interfering expressions can be evaluated in parallel. The FX programming language was a remarkable demonstration of the possibilities brought by type-and-effect systems, but the lan-

guage lacks a complete type inference algorithm. The need to provide type and effect annotations is burdensome, and makes the language unsuitable for mainstream use and adoption.

Talpin and Jouvelot developed a complete type reconstruction algorithm for a polymorphic type-and-effect system with subeffecting [TJ92]. The notion of subeffecting is similar to that of subtyping, and it is required to handle certain uses of first-class functions. Remarkably, theirs is a constraint-based type inference algorithm, essentially an extension of Damas-Milner's *Algorithm W* [DM82]. (Sub-effecting is translated into a system of inequalities that restrict the lower bounds of effect variables.) It requires no annotations and infers, for each expression, its most-general type, and its minimal set of effects. A major limitation of their approach is that, while efficient at inferring effects, it is inappropriate for *restricting* the effects of expressions—I will explain why this is useful in the following.

Koka [Lei14] is a functional programming language by Daan Leijen, featuring row-polymorphic [Rem93] effect types. By modeling sets of effects as *rows of effects*, Koka effectively supports effect polymorphism and a form of subeffecting. Koka is not only concerned about inferring the effects associated with the evaluation of expressions. It also allows language designers, or even users, to specify upper bounds for the effects of certain expressions, enabling the type-checker to catch more errors. For instance, expressions used in *assert* constructs should not have side-effects, so that asserts can be removed without altering the behavior of the program. Unlike in Talpin-Jouvelot's effect system, in Koka it is possible to enforce such a constraint.

In my bug-finder, EBA, the effect system is an internal tool, used to determine whether an expression performs certain operations on a set of resources of interest. But, unlike optimizing compilers, that essentially track reads and writes to memory locations, EBA relies on user-definable effects to track arbitrary operations on resources. Complete type inference is a must here, otherwise the annotation burden would make EBA unusable. For that reason, I have settled on Talpin-Jouvelot's type-and-effect system as a basis. Their inference algorithm is efficient, requires no annotations, and supports first-class functions—thereby, also C function pointers. Note that Damas-Milner type inference performs well in practice [McA96], and lower-bound subeffecting constraints can be solved in polynomial time.

2.2.2 Pointer analysis

Side-effect analysis tracks operations on resources, which are identified by their location in memory. But, in most programming languages, the exact location of data in memory is only known at runtime. For instance, two `FILE` pointers f_1 and f_2 may, or may not, refer to different `FILE` objects in memory. Expressions that denote the same memory location are called *aliases*. Aliasing can make the meaning of programs, like `fclose(`f_1`); fclose(`f_2`);`, to change radically, from being a legal program that closes two files, to undefined behavior, if f_1 and f_2 refer to the same file descriptor. Pointer analyses are then used to compute a set of potential values for a pointer expression at compile-time [SB15]. Naturally, side-effect analysis embeds pointer analysis.

The most popular pointer analyses in the literature are those that compute *points-to* graphs. For instance, given the assignment `p = &x`, these analyses record that `p` points-to `x`. Expressions are consider to alias if their points-to sets intersect. For a programming language like C, that allows liberal use of pointers and type casts, computing accurate points-to information is particularly expensive. Unsurprisingly, the classic and most popular pointer analyses are whole-program context- and flow-insensitive analyses [And94, Ste96b, Das00]. Modular points-to analysis is still an active research topic [SB15], and whole-program flow-sensitive points-to analysis does not scale to real programs (let alone Linux!).

Side-effect analysis is not so much interested in points-to sets, as it is in aliasing relations. Given the assignment `p = &x`, an alias analysis records that `*p` and `x` are aliases, i.e. denote the same object in memory. Alias analyses can use representations that are more amenable to decomposition than a points-to graph. In fact, type-and-effect inference typically performs polymorphic and modular alias analysis [TJ92, Lei14]. These type-and-effect systems introduce the concept of *memory region*, that abstracts sets of possibly aliased memory locations. Types are annotated with the regions where objects are stored in memory, and aliasing relations are recorded by unification. This is similar to Steensgaard's points-to analysis [Ste96b], but it is significantly more precise thanks to region polymorphism [FFA00, Hin01].

Among the many trade-offs to consider in performing pointer analysis, the treatment of records (C structures) is one of the most important in practice [Ste96a, YHR99]. The simplistic approach consists in "collapsing" record fields, so that only aliasing relations between record objects, and not between their individual fields, are considered [YHR99]. Yet,

real software makes extensive use of records, and ignoring their structure yields poor results in practice. Records are used to describe complex resources, and each field constitutes a sub-resource that also needs to be tracked individually.

Unfortunately, in C, type casts between structure types make alias analysis for individual record fields non-trivial. For instance, given a pointer `s` to an `struct A`, what does `((struct B *) s)->x` mean? The answer depends on the memory layouts of `struct A` and `struct B`, which are largely implementation dependent. Many C projects, including Linux, do exploit type conversions between structure types, to implement OO-like abstract interfaces and class hierarchies (for instance 7acf6cd80b2, cf. Sect. 3.7). The Linux device driver model is a good example of this.

If alias analysis must be sound, there are situations where it is necessary to partially collapse record fields [Ste96a, YHR99]. Fortunately, for the purpose of bug finding, we are allowed to sacrifice soundness to gain in precision [LSS+15]. (Unlike for formal verification, where we require a proof of correctness, we can accept that bug finders may miss bugs due to unsoundness.) My treatment of record types is a relaxed version of the *common initial sequence* approach described in [YHR99], which captures many common uses of type casts between structure types (cf. Chapter 4).

2.3 Static resource safety

In this section, I discuss the state of the art techniques and tools for automatically checking resource manipulation safety at compile-time, and how the bug finding technique of Chapter 6 relates to them. I have classified the many available techniques in four groups, according to their degree of precision and scalability: static typing (Sect. 2.3.1), static code checking (Sect. 2.3.2), heavyweight static analysis (Sect. 2.3.3), and software model checking (Sect. 2.3.4).

2.3.1 Type-safe resource manipulation

Many works have developed approaches to type-safe resource manipulation, for instance [SY86, FTA02, PFH06, FJKA06, KS08]. These techniques impose stricter typing disciplines that statically check that operations are applied on resources, only in the contexts where it is legal. For

instance, a read operation on a file, that *might* not be open at a certain program location, is rejected by the type-checker.

The simplest approaches to type-safe resource manipulation are still flow-insensitive analyses, as most type systems. In [KS08], Kyselyoiv and Shan present a technique to statically ensure the safe use of file handles (and potentially other kinds of resources) using regions [TT94]. Their approach requires a quite sophisticated underlying type system, that provides features such as rank-2 polymorphism [JVWS07]. In [FFA99], Foster et al. extend the C type system with user-defined type qualifiers, and provide a constraint-based inference algorithm to check for qualifier inconsistencies. Type qualifier inference can be used, for instance, to perform flow-insensitive taint analysis [JW04, BHS03]. They realized this idea into a tool, CQual [FJKA06], that has found a number of bugs in Linux, nonetheless it produces an overwhelming amount of false alarms [JW04].

On the other hand, the more advanced approaches are based on *flow typing* [Pea13]. One of the earliest of these works is [SY86], where Strom and Yemini introduce the concept of *typestate*. Typestate is a flow-sensitive (but path-insensitive) abstraction of the state of an object, that determines which operations are permitted at a specific program location. Operations must be annotated with a typestate precondition (e.g., to write a file this needs to be open for writing), and one or more postconditions that define *typestate transitions* (e.g., the `fclose` operation turns an open file into a closed file). In essence, this is analogous to associate each resource type with a state machine description. However, compile-time *typestate tracking* is limited to programming languages with restricted use of pointers (so that two different names cannot denote the same run-time object), and with a concurrency model based on message passing (so that two different threads cannot share data). This is certainly not the case of C, in which I am interested.

In [FTA02], Foster et al. take [FFA99] a step further, and extend their previous work (and their tool, CQual) with flow-sensitive type qualifiers. This notion of qualifier, that now tracks the state of objects, is basically the same as the concept of typestate introduced in [SY86]. Yet, type qualifiers are more expressive than typestate, and admit subtyping (e.g., a file that is open in `readwrite` mode is also open in `read` mode). This work also considers the many intricacies of a language like C although, due to the lack of shape polymorphism in their system, generic functions that operate on `void` pointers are handled by implementation tricks.

(In EBA, these functions would receive shape-polymorphic signatures.) The authors use CQual to verify lock management in various Linux device drivers, and find a number of bugs among a vast amount of false alarms. In addition, for CQual to be run on Linux, they had to alter the source code in many non-trivial ways, including the addition of CQual-specific annotations.

Similar to CQual, Locksmith is a verification tool for detecting data races in C programs, that builds upon constrained-based type inference [PFH06]. Locksmith automatically infers the set of locks that protect each concurrent access to a memory location. A program is considered free of data races, if every memory location is consistently protected by the same set of locks. Locksmith requires few annotations, and handles non-lexically scoped locks. While it can deal with unrestricted aliasing of memory locations—at the cost of plenty of false positives, Locksmith does require that every lock variable can be mapped to a unique lock object at run-time. This linearity condition cannot easily be enforced when locks are part of data structures, or are involved in non-trivial type conversions, such as in the Linux kernel—I will show examples of such problems in Chapter 3.

Approaches to type-safe resource manipulation are interesting additions to consider when designing new programming languages. Some of these techniques have, in fact, been considered in the development of cutting-edge programming languages, such as Rust.[3] It is less clear that these approaches can be effectively integrated into programming languages like C, that have not been designed with compile-time safety in mind. Attempts at integrating these techniques into C are largely unusable in practice due to the noise caused by false positives, and the burden of rewriting code or adding annotations to remove these [FTA02, PFH06]. Not least, such rigid typing disciplines would be hardly adopted by some C programmers, especially in the operating systems domain. EBA builds on top of a type-and-effect analysis, but uses the inferred effect information merely as an abstraction—as opposed to an specification, on which to find bug patterns (cf. Sect. 2.3.2).

2.3.2 Static code checking

Static code checkers (also known as code scanners, or linters) are lightweight bug-finding tools. Unlike heavyweight static analyzers and

[3] https://www.rust-lang.org/

software model checkers (cf. sections 2.3.3 and 2.3.4), these are mostly syntax-based tools that know little about program semantics. They ignore aliasing, rely on heuristics, do not compute summaries, and work intra-procedurally. With a focus on pragmatism, they analyze incomplete source code, sometimes even CPP code, and try to report as fewer false bugs as possible, at the cost of being *unsound*. Hence, static checkers are easy to set up and use, run fast, and scale well. But, for the same reasons, they find relatively simple and shallow bugs.

The degree of sophistication of these tools varies significantly. The simplest ones would not even fully-parse the source code [Kop97, BNE16], whereas others check finite-state properties against the control-flow of programs [ECCH00]. To my knowledge, the first tool of this class was lint [Joh78, Dar86]—hence the name *linters*, a static checker for C created by Stephen C. Johnson at Bell Labs in 1970s. Lint will report *probably* buggy code and bad coding style, and make portability recommendations if desired. For instance, the statement `*p++;` is equivalent to `*(p++);`, and therefore the pointer dereference (i.e., the `*`) has no purpose. This may be a bug if the intention was to write `(*p)++;` instead.

Splint [EL02] can be regarded as a modernized and more sophisticated version of lint which, in addition, exploits user-provided code annotations. CppCheck[4] supports C and C++, and is able to analyze the different compile-time alternatives of CPP `#if` conditionals. It is, to my knowledge, the only production-ready variability-aware bug finder available (cf. Section 1.1). CppCheck is a good example of the philosophy behind bug finders: it employs a partial preprocessor [KKHL10], a fuzzy parser, a simple value analysis, and finds bugs matching patterns at the level of token. There are many of these tools available that will not be discussed here. Nonetheless, three of them deserve special attention, for having been adopted to analyze the Linux kernel source code [CWY+13]. These are Sparse, Coccinelle, and Smatch.

Sparse[5] exploits the GCC *attributes* extension to define Linux-specific annotations. These annotations are essentially type qualifiers, and Sparse acts like a sort of type-checker by ensuring that their semantics is respected. For instance, the `__user` and `__kernel` annotations are used to specify whether a pointer belongs to user or kernel space, respectively. User and kernel pointers are treated as having incompatible

[4] `http://cppcheck.sourceforge.net`
[5] `http://sparse.wiki.kernel.org`

types [BA08]. Sparse will also check that user pointers are not dereferenced directly by kernel code, for security and other reasons. Among other kinds, it supports three function-level locking annotations: `__must_hold`, `__acquires`, and `__releases`. Sparse will detect errors such as the release of a lock followed by a call to a function that requires the caller to hold that same lock. The main drawback of Sparse is that it makes no effort at inferring these annotations. Hence, locking checks are unpopular among developers because the annotation burden is too high—a large number of functions in Linux will operate directly *or indirectly* on locks [XA05].

Coccinelle[6] is a program transformation tool, with an associated domain-specific language (DSL) to specify flow-based transformations [BDH+]. These transformations, called *semantic patches*, resemble diff-like patches with the addition of *metavariables*, to abstract away variable and function names, and a wildcard pattern ("..."), to skip uninteresting code blocks. While originally conceived for managing collateral evolution of code [PLM06], Coccinelle flow-matching capabilities have been used to encode bug-finding rules [LLH+10, PTS+11]. The Linux kernel sources include a number of Coccinelle-based bug checkers, located in `scripts/coccinelle/`, which can be run with `make coccicheck`.

Smatch[7] realizes the idea of *meta-level compilation* proposed by Engler et al. [ECCH00]. It is, essentially, a scriptable intra-procedural data-flow analysis engine. A checker in Smatch defines a state machine, where transitions are associated with specific operations: e.g., a call to `spin_lock` will change the state of a spin lock object from *unlocked* to *locked*. A bug is reported whenever an object transits to an error state; for instance, due to a `spin_lock` call on an already *locked* spin lock. It reports a low number of false positives thanks to Linux-tailored triaging heuristics, and to a limited yet effective tracking of path correlations. Smatch warnings have led to thousands of bug-fixing commits in Linux.

EBA, the tool proposed in this project, is also a bug finder. But, unlike the aforementioned tools, it is able to track manipulation of resources across functions boundaries. EBA scales well and runs fast and yet, it finds deep inter-procedural bugs that the other bug finders do not (cf. Chapter 7). This is achieved thanks to the use of lightweight, and computationally cheap, shape-and-effect abstractions (cf. Chapter 4). Sparse will miss the bug of Fig. 1.2, because the functions involved have no

[6] `http://coccinelle.lip6.fr`
[7] `http://smatch.sf.net`

locking annotations. (Linux developers rarely add Sparse locking annotations.) It is also missed by Smatch, and by the Coccinelle double-lock script that is shipped with the Linux sources. This is because the second acquisition of the lock happens through a nested call to function `inode_get_rsv_space`, something EBA knows looking at its effect signature (cf. Fig. 1.3).

It is worth noting that there are workarounds to add some inter-procedural capabilities to otherwise intraprocedural bug finders. In fact, Smatch has limited support for inter-procedural bug finding. As in [ECCH00], Smatch provides a script to traverse the whole program and collect all functions that, for instance, may perform locking. On a regular laptop, this script takes a few hours to run on the entire Linux kernel, and must be run n times to examine n levels of the call graph. Compared to a proper effect system such ad-hoc scripts are difficult to extend; and do not track aliasing, nor handle function pointers appropriately.

2.3.3 Static program analysis

Static program analyzers have a good understanding of program semantics. Unlike static checkers, these tools are designed to *prove* the absence of certain types of bugs. In practice, though, some trade-offs are required [LSS+15], and most of these tools have at least a few sources of unsoundness (e.g., GCC inlined assembly). These analyses can track the value of expressions at each program location. Consequently, static analyzers are able to find complex and deep bugs, even when these involve non-trivial data dependencies. In order to scale, they rely heavily on *abstraction* to model the program state. There is no single abstraction that works for every problem, and tools use one abstract domain or another according to their target class of bugs. The use of coarse abstractions can make these tools report a considerable amount of false positives, whereas very precise abstractions are expensive to compute.

Astrée is a whole-program analyzer that is designed to show the absence of runtime errors in mission-critical embedded C software [CCF+09]. It detects any potential violation of the C ISO/IEC 9899:1999 standard (division by zero, null-pointer dereference, etc.), implementation-specific limits (e.g., arithmetic overflows), and user-provided assertions. It is famous for having proven the absence of runtime errors in the flight control software of some Airbus airplane models. Astrée is particularly good at understanding arithmetic, for what it uses

a combination of abstract domains [CC77, Cou96]. For instance, if the value of an expression is approximated by the interval $[-1, 1]$, its use as a divisor will produce a warning about a potential division by zero (because $0 \in [-1, 1]$). (Note that the interval is an over-approximation, and the actual value of the expression at runtime may, in fact, never be zero.) Astrée targets a very specific kind of software (*command/control software*), and only supports a restricted subset of the C language. This language subset excludes recursion, dynamic memory, concurrency, etc. Astrée scales up to code bases of hundreds of thousands lines of code, but its running time is typically measured in hours.

Data-flow analyses compute facts about the program state at each source location [NNH99]. An example of data-flow fact is, for instance, whether a variable has been initialized or not. These facts are much simpler than the abstract domains employed by Astrée. Compiler warnings are good examples of static analysis based on tracking data-flow facts: e.g., the compiler warns whenever a variable, that may not have been initialized, is used. For efficiency, compilers implement intra-procedural analyses, and their warnings can be overly conservative at times. In [RHS95] Reps, Horwitz and Sagiv show how an important class of inter-procedural data-flow analyses, coined as *IFDS problems*, can be performed precisely and efficiently. This is thanks to an efficient representation of partial function summaries. Their algorithm, based on graph reachability, is popularly known as *RHS*. RHS has been the basis for many interprocedural static analyzers. The Clang Static Analyzer[8], for instance, has recently gained interprocedural analysis capabilities by using an adaptation of this technique [SYRS16].

INFER [CD11] is an automated verification tool based on separation logic [BTSR04]. (It is an industrial-strength development of the research prototype Smallfoot [BCO05].) At the present time, it is developed at Facebook, where it is used to find specific kinds of memory and resource manipulation errors in their mobile applications. INFER was originally designed to prove the absence of memory errors in C programs, for which it performs (*deep-heap*) shape analysis [SRW98]. It is capable of reasoning about complex dynamic data structures, such as circular and nested double-linked lists. INFER scales to arbitrarily large code bases, and has run on projects with millions of lines of code, such as the Linux kernel. For this, INFER synthesizes function specifications, in the form of Hoare triples that entail memory safety, using *bi-abduction inference* [CDOY11]

[8] `http://clang-analyzer.llvm.org`

Such specifications are used as summaries to perform compositional inter-procedural shape analysis [DOY06].

EBA offers a compromise solution between code scanners (cf. Sect. 2.3.2) and static analyzers. Code scanners search for syntactic patterns, scale well and are fast, but cannot find complex nor deep bugs; whereas static analyzers rely on abstract semantics, and can find complex and deep bugs, but in turn are slow. EBA employs a lightweight and cheap abstraction based on computational effects. It knows which operations are executed at each program location, but it has a shallow understanding of the heap and how the data flows. EBA is well suited to find conceptually simple, but interprocedurally deep bugs, when these can be understood as illegal sequences of operations, and do not involve complex data dependencies—i.e., "resource mis-manipulation bugs". It is moderately slower than most linters, but orders of magnitude faster than most static analyzers. For instance, calling twice a procedure that should only be called once, may result in a kernel panic[9]. Finding a legal execution trace that leads to a call to `panic()` is a hard problem, yet the high-level bug pattern is trivially specified in EBA.

2.3.4 Software model checking

Software model checking (SMC) describes a range of techniques to systematically explore the state space of a software program. These are also verification tools, capable of finding deep and even more complex bugs than static analyzers (cf. Sect. 2.3.3). Unlike static analyzers, which employ abstractions that combine the analysis of multiple program paths, SMC tools analyze each individual execution path. Hence, these tools have to deal with the *exponential path explosion* problem, in addition to the *state-space explosion* problem. SMC tools have bigger scalability problems than static analyzers, but can be more precise at tracking the value of expressions. Typically they represent the program state symbolically, as logical formulas, rather than abstractly. Symbolic representations are extremely precise, but require the use of automated theorem provers, which adds significant computational cost.

Bounded model checking (BMC) tackles the state- and path explosion problem by bounding both the size of the state, and the depth of the paths to analyze. Typical BMC techniques verify programs by translating them into satisfiability problems: programs are encoded into some

[9] `http://vbdb.itu.dk/#bug/linux/472a474`

(semi-)decidable logic, and a SAT [BBH+09] solver is used to explore the state space. Among other examples of BMC tools for C, CBMC [CKL04] encodes programs at the bit-level, into Boolean propositional logic. For such encodings to be possible, loops must be completely unrolled, and function calls must be fully inlined. Also, there may be C constructs that cannot be adequately modeled in the underlying logic. Yet, BMC is capable of verifying the absence of runtime errors in certain types of embedded software, that have statically known bounds. When loop and recursion bounds are unknown at compile-time, or when programs are large, BMC tools can be used as bug finders by exploring execution paths up to a given depth. As bug finders, BMC tools are much more precise, but scale much worse, than the tools discussed in Section 2.3.2.

Symbolic executors exhaustively explore all the paths in a program, akin to symbolic model checkers. The program state is maintained symbolically for each program path. If, for instance, a pointer dereference is reached, the executor checks whether the value of the pointer is a valid memory address. This check consists in one or more queries to a SAT solver. If the bug is confirmed, the executor is able to produce a test case; i.e., it provides concrete program inputs that trigger the bug. An example of such tool is KLEE [CDE08], essentially a *symbolic* virtual machine built on top of LLVM [LA04]. In practice, tools like KLEE introduce certain restrictions to bound the search space; and perform many kinds of optimizations. Nonetheless, KLEE does not scale to large and complex software like the Linux kernel.

Another kind of SMC deals with the state-space explosion by means of *predicate abstraction* [BMMR01]. These SMC tools consist of an abstraction-check-refinement loop. The target program is abstracted in terms of a set of predicates, resulting in a Boolean program [BMMR01]. For instance, given a single predicate `x` $= 0$, both a statement `x = 1;`, and a condition `x > 1`, would be abstracted as `x` $\neq 0$. This Boolean program is model-checked in search of bugs [BR01a]. If no bug is found, the program can be considered "correct", given that—at least in theory—the Boolean abstraction accepts all the behaviors of the original program. If a bug is found, and the execution path that leads to it is feasible, the SMC tool provides a test case. If otherwise a false positive is found, it is possible to use *interpolation* [McM05] to refine the set of predicates and restart the process [CGJ+00]. This technique can work well for programs that are control dominated, such as device drivers. But, if the program contains non-trivial data dependencies, the refinement loop can

take too many iterations, or even diverge. Examples of SMC tools are BLAST [BHJM07], used by the *Linux Driver Verification Project* [ZMM+15], and SLAM [BR01b], developed and used at Microsoft to statically analyze Windows device drivers.

EBA can be seen as a model checker based on an abstraction-check-refinement loop. In a first step, it builds an effect-based abstraction of the program to analyze. Subsequently, it performs a depth-first search of the effect-annotated control-flow graph of each function. During this search, EBA tries to find illegal sequences of operations (i.e., "bug patterns"), such as the double-locking of a non-reentrant lock. EBA scales because path exploration is, most of the time, confined to a single function. The majority of function calls will be deemed irrelevant by simply looking at their effect signature (e.g., the function does not operate any lock). If a function call does seem relevant but, due to the imprecision of the abstraction, its effect signature does not allow to draw any conclusion, then the function call is inlined and the search resumed. In this context, function inlining can be understood as a form of abstraction refinement.

Chapter 3

A Qualitative Study of Bugs in Linux

Originally published in: ASE 2014[1]

While static analysis techniques and tools abound, it seems that, in comparison, little effort has been put into understanding the characteristics of bugs in large software systems. Gaining such understanding is needed to ground research in actual problems. When I started this PhD project, I could only find a handful of studies of software bugs in the Linux kernel (cf. Section 2.1). These studies, so far, have focused on analyzing the frequency of a narrow set of pre-defined types of bugs in Linux [CYC+01, PTS+11]. The question is whether we have a good perspective of what bugs Linux developers actually spend time fixing.

In the context of the VARIETE project, we are particularly interested in studying the occurrence of bugs in highly-configurable software (cf. Section 1.1). It turns out that there is also little knowledge about the characteristics of software bugs in Linux, or in any other configurable system. (Yet, there is a plethora of variability-aware extensions to static analysis and model-checking algorithms!) Given that Linux supports about two dozen architectures, and is parameterized by tens of thousands of configuration options, it seems reasonable to assume that it suffers from numerous variability bugs. But, while bug reports abound, it is unclear how many of those bugs are caused by feature interactions.

I set off to gain understanding on the complexity and nature of (variability and feature-interaction) bugs in the Linuxkernel. I have started to

[1]This chapter corresponds to our paper entitled *"42 Variability Bugs in the Linux Kernel: A Qualitative Analysis"*[ABW14], published in the 29th International Conference on Automated Software Engineering (ASE 2014).

approach this objective via a qualitative in-depth analysis, and documentation, of 43 cases of such bugs. This study makes the following contributions:

- *Identification of 43 variability bugs in the* Linux *kernel*, including in-depth analysis and presentation for non-experts.
- *A database containing the results of the analysis*, encompassing a detailed data record about each bug. These bugs comprise common types of errors in C software, and cover different types of feature interactions. The current version is available at http://VBDb.itu.dk/.
- *Self-contained simplified versions of all bugs.* These ease comprehension of the underlying causes, and can be used for testing bug-finders in a smaller scale.
- *An aggregated reflection over the collection of bugs.* Providing insight on the nature of bugs in a large project like Linux.

This work is directed to designers of program analysis and bug finding tools, like myself. I believe that this collection of bugs can inspire others, as it inspired me, in several ways: (i) it provides a set of concrete and well described challenges for analyses; (ii) it supports the evaluation of new techniques at design stage, since they can be tried on simplified Linux-independent bugs; and (iii) it serves as a benchmark for evaluating our tools.

3.1 Background

The concepts of configurable software, feature, and variability bug have been introduced in Section 1.1. This section introduces the concepts of *feature-interaction bugs* (Sect. 3.1.1), and *variability bug fixes* (Sect. 3.1.2).

3.1.1 Feature-interaction bugs

Figure 3.1 presents a tiny preprocessor-based C program family using two features, INC and DEC, that contains a bug. Note that statements at lines 10 and 13 are conditionally present. The *presence condition* of a code fragment is a *minimal* (by the number of referred variables) Boolean formula over features, specifying the subset of configurations in which the

```
1  int printf(const char * format, ...);
2
3  void foo(int a) {
4          printf("%d\n",42/a); // ERROR
5  }
6
7  int main(void) {
8          int x = 1;
9          #ifdef CONFIG_INC    // DISABLED
10         x = x + 1;
11         #endif
12         #ifdef CONFIG_DEC    // ENABLED
13         x = x - 1;
14         #endif
15         foo(x);
16 }
```

Figure 3.1: Example of a program family and a bug.

code is included in the compilation. For instance, the assignment in line 10 has presence condition INC, this statement is therefore present in configurations INC $\wedge$ DEC and INC $\wedge \neg$DEC. The concept of presence condition extends naturally to bugs, that is, the subset of configurations in which a bug occurs.

If features INC and DEC can be independently set, the program family of Fig. 3.1 defines four different program variants. But more often than not features would have interdependencies. This may be because one feature relies on another, or because two features conflict and cannot be simultaneously enabled, and so on. For instance, in Linux, support for the *ecryptfs* file system (feature ECRYPT_FS) requires the inclusion of the cryptographic API (feature CRYPTO). Feature dependencies are specified using a *feature model*, essentially a propositional formula over features constraining legal configurations. The Kconfig language, which is part of the Kernel build (Kbuild) system, is used to specify the feature model of the Linux kernel.[2]

Features can also, intentionally or not, influence the functions offered by other features, a phenomenon known as *feature interaction*. In

[2] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

Figure 3.1, the two features interact because both modify the same program variable `x`. Enabling either INC or DEC, or both, results in different values of `x` prior to calling `foo`. If INC is disabled and DEC is enabled, this interaction causes a crash at line 4, when the program attempts to divide by zero. Bugs that are caused by the (often unexpected) interaction of two or more features are called *feature-interaction bugs*. The *degree* of a feature-interaction bug counts the number of features involved in the interaction. Our example division-by-zero bug is thus a 2-degree bug with presence condition $\neg$INC $\wedge$ DEC.

3.1.2 Variability bug fixes

The Linux kernel, as a program family, is stratified in three layers: the Kconfig feature *model*, the C *code*, and the *mapping* of features to code. In Linux, this mapping is specified using CPP conditionals (`#if` and so on), and also in Makefiles. In the following, I will show how our example division-by-zero bug could, depending on the interpretation, be assigned to and fixed in each of these layers separately.

Fix in code. If function `foo` should accept any `int` value, then it may be that the dividend and divisor were erroneously inverted. The following patch fixes the problem:

```
1 @@ -4 +4 @@
2 -       printf("%d\n",42/a);
3 +       printf("%d\n",a/42);
```

Fix in mapping. If function `foo` was not supposed to handle a zero input, then it may be that the original intention was to decrement `x` only when both features are enabled. The following patch fixes the problem:

```
1 @@ -12 +12 @@
2 -  #ifdef CONFIG_DEC
3 +  #if defined(CONFIG_DEC) && defined(CONFIG_INC)
```

Fix in model. Alternatively, if `foo` does not accept zero as input, another possibility is that feature DEC should depend on INC, and that dependency is missing in the feature model. The following Kconfig model fixes the problem:

```
1 config INC
2         bool "Increment variable x"
3
4 config DEC
5         bool "Decrement variable x"
6         depends on INC
```

3.2 Methodology

3.2.1 Objective

My objective was to qualitatively understand the complexity and nature of bugs in highly-configurable systems-level software (cf. Section 1.3).

This includes addressing the following research questions:

RQ1 What are the variability characteristics of bugs?

RQ2 What are the challenges for program analysis tools?

RQ3 What kind of bug finders do we need?

3.2.2 Subject

I study Linux, which is likely the largest highly configurable open-source system in existence, with more than 14 million lines of code, and 16 thousand configuration options, as of 2016. Crucially, I have free access to the bug tracker[3], the source code and change history[4], and to public discussions on the mailing list[5] and other forums. There also exist books on Linux development [BC05, Lov10], which are valuable resources when understanding a bug-fix.

[3] https://bugzilla.kernel.org/
[4] http://git.kernel.org/
[5] https://lkml.org/

Message filters
`CONFIG_`*fid*
`configuration`
`config option`
`if` *fid* `is [not]? set`
`when` *fid* `is [not]? set`
`if` *fid* `is [en|dis]abled`
`when` *fid* `is [en|dis]abled`

Content filters
`select` *fid*
`config` *fid*
`depends on` *fid*
`#if`
`#ifdef` *fid*
`#else`
`#elif`
`#endif`

Figure 3.2: Regular expressions selecting configuration-related commits; where *fid* abbreviates `[A-Z0-9_]+`, matching feature identifiers.

3.2.3 Part 1: Finding variability bugs

I focus my attention on bugs already found, corrected, and merged into the stable branch of Linux. These bugs have been publicly discussed (usually on LKML) and confirmed as actual bugs by the developers, so the information about the nature of the bug fix is reliable, and I minimize the chance of including fictitious problems. In early 2014, when this study was conducted, the Linux stable repository had over 400,000 commits. This large commit history rules out manual investigation of each commit. Thus, I have settled on a semi-automated search through Linux commits to find (variability) bugs via historic bug fixes, using the following steps:

1. *Selecting variability-related commits.* I find commits whose *message* indicates a variability-related change; or whose *patch* appears to alter the feature model, the feature mapping, or configuration-dependent code. This is achieved by matching case insensitively the regular expressions of Figure 3.2. Expressions on the left, identify commits in which the author's *message* relates the commit to specific features. Those on the right, identify commits introducing changes to the Kconfig feature model, the (CPP) feature mapping, or code near an `#if` conditional. This step selects in the order of tens of thousands of commits.

2. *Selecting bug-fixing commits.* I further narrow to commits that potentially fix bugs and thus, together with the previous filter, we obtain candidates for variability bug-fixes. This is achieved by

Generic filters
`bug`
`fix[es]`
`closes \#`
`oops`
`warn`
`error`
`unsafe`
`invalid`
`violation`
`end trace`
`kernel panic`

Specific filters
`void \*`
`unused`
`overflow`
`undefined`
`deadlock`
`double lock`
`memory leak`
`uninitialized`
`dangling pointer`
`null [pointer]? dereference`
`...`

Figure 3.3: Regular expressions selecting bug-fixing commits.

matching (also case insensitively) the regular expressions of Figure 3.3 against the commit message. Expressions on the left are generic keywords that can appear in any bug-fixing commit message. Expressions on the right of the figure try to identify bug-fixing commits for specific types of bugs, such as void-pointer dereferences (`void *`), undefined symbols (`undefined`), use before initialization (`uninitialized`), and so on. Generic filters may select thousands of commits in Linux, whereas specific filters may select only a few hundreds or tens.

3. *Manual scrutiny.* Finally, I read the commit message or the issue, and inspect the changes introduced by the commit to remove false positives. For instance, commit `7518b5890d` passes through the previous two filtering steps yet, after examining the complete commit message, it became clear that it does not fix a bug, but adds new functionality. I examined simple commits first. A complex commit either introduces more than a few changes (I choose a cut-off value of *ten*), or affects very complex subsystems (e.g., the *kernel/sched* Linux subsystem). The ideal commit has an elaborated *message* providing some form of error trace, and introduces few modifications.

The selection of the keywords and patterns of figures 3.2 and 3.3 was based on my own understanding of the Linux kernel, and a preliminary analysis of Linux code and commit messages to identify keywords and phrases used by developers. For instance, I learned that, in Linux, an

Oops refers to a critical condition detected by the kernel, such as a `NULL` pointer dereference in kernel code.

3.2.4 Part 2: Analysis of bug candidates

The second part of the methodology is significantly more laborious than the first part. For each variability bug identified, I manually analyze the commit message, the patch fix, and the actual code to build an understanding of the bug. When more context is required, I find and follow the associated discussion on LKML. Code inspection is supported by ctags,[6] and the Unix grep utility.

1. *The semantics of the bug.* For each bug, I want to understand the *cause* of the bug, the *effect* on the program semantics, and the relation between the two. This often requires understanding the inner workings of the kernel, and translating this understanding to general programming language terms accessible to a broader audience. As part of this process, I try to identify a relevant *bug trace*, and collect links to available information about the bug online.

2. *Variability related properties.* I establish what is the presence condition of a bug (precondition in terms of configuration choices), and where it was fixed: in the code, in the feature model, or in the mapping.

3. *Simplified version.* I condense my understanding in a *simplified version of the bug*. This serves to explain the original bug, and constitutes an easily accessible benchmark for testing and evaluating ideas and proof-of-concept prototypes.

I analyzed bugs from the previous step (cf. Section 3.2.3) following this method, and stored the reports from this analysis in a publicly available database. The detailed content of the report is explained in Section 3.4.

3.2.5 Part 3: Data analysis

I reflect on the set of collected data in order to find answers to my three research questions (cf. Section 3.2.1). This step is supported with some

[6] `http://ctags.sourceforge.net/`

quantitative data but, importantly, I do not make any quantitative conclusions about the population of bugs in Linux (such conclusions would be unsound given the above research method). The analysis purely characterizes diversity of the data set obtained. It allows me to present the entire collection of bugs in an aggregated fashion (see sections 3.5, 3.6, and 3.7). The qualitative analysis of the bugs suggests directions for developing tools. Finally, in order to reduce bias I confronted this method, findings, and hypotheses in an interview with a professional Linux Kernel Engineer.

3.3 Threats to validity

3.3.1 Bias due to the selection process

Given that I extract bugs from commits, the bug collection is biased towards bugs that are easier to reproduce—thus more likely to be found and get fixed. Bugs that occur under very exceptional circumstances, or that have no visible effect from the user perspective—like many security flaws, can long remain unnoticed. Also, since users run a small subset of all the possible configurations of Linux, there are potentially many unknown variability bugs.

Further, the success of a keyword-based search relies on the ability (and willingness) of developers to properly describe bugs and identify the affected configurations. I have mitigated this bias by searching for generic bug-keywords (e.g. commit f7ab9b407b3 was caught this way), and by searching for variability-related keywords not only in the commit message but also in the patch fix (e.g. commit 76baeebf7df).

In order to minimize the risk of introducing false positives, I do not record bugs if I fail to extract a sensible error trace, or if my understanding of the bug does not match with the commit description. This may introduce bias towards reproducible and lower complexity bugs. In spite of that, Section 3.7 shows that the bug collection contains many complex cases, such as 7acf6cd80b2.

Because of inherent bias of a detailed qualitative analysis method, I am not able to make quantitative observations about bug frequencies and properties of the entire population of bugs in Linux. Note, however, that we are able to make qualitative observations, as well as formulate hypotheses, from this data. Interestingly though, the bug collection still exhibits very wide diversity as shown in Section 3.5.

3.3.2 False positives and overall correctness

The analysis of the bugs is not run by a domain expert, which introduces the risk of mistaken identification of bugs. By only considering bugs that have been identified and fixed by the developers, I mitigate the risk of introducing false positives. I only take bug-fixing commits from the Linux stable repository, which have been extensively reviewed.

In addition, the data can be independently verified since it is publicly available, and easily accessible through a web interface. Since the publication of this study in 2014, many researchers have analyzed and used this data, and some minor problems have been reported and fixed.

The risk of introducing false positives is not zero though, it also occurs that developers conservatively fix non-bugs. For instance, Linux commit b1cc4c55c69 adds a nullity check for a pointer that is guaranteed not to be `NULL`.[7] It would be tempting to record it as a bug fix, while in fact it adds a defensive check to handle a *potential* bug.

The manual analysis of a bug to extract a bug trace is also error prone, especially for a complex system like Linux. (This also applies to the derivation of simplified bugs.) Ideally, I would have supported this manual analysis with a program slicing tool, if any such tool, that scales up to Linux and ideally copes with variability, existed and were freely available.

3.3.3 External validity

The focus on a single software project, and the size of the bug collection, do not allow to generalize the observations derived from this study. The process of collecting and analyzing these 42 bugs took several months of work. It is infeasible to study a significantly larger number of bugs using this method. Nonetheless, I believe that the importance of the Linux kernel itself justifies a wide and deep investigation of its errors, even if it limits generalizability.

To my knowledge, there was no better way of answering my research questions. A more automated approach based on bug-finders would not be satisfactory. Bug-finders search for certain classes of errors, so they can give good statistical coverage for those classes, but they would not be able to assess the diversity of bugs that appear.

[7] `https://lkml.org/lkml/2010/10/15/30`

```
                                - 6252547b8a7 -

type:    Null-pointer dereference

descr:  Null pointer on !OF_IRQ gets dereferenced if IRQ_DOMAIN

        In TWL4030 driver, attempt to register an IRQ domain
        with a NULL ops structure: ops is de-referenced when
        registering an IRQ domain, but this field is only set
        to a non-null value when OF_IRQ.

config: TWL4030_CORE && !OF_IRQ

bugfix:

    repo:  git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git

    hash:  6252547b8a7acced581b649af4ebf6d65f63a34b

    layer: model, mapping

trace:
     .  dyn-call drivers/mfd/twl-core.c:1190:twl_probe()
     .  1235:  irq_domain_add(&domain);
     ..  call kernel/irq/irqdomain.c:20:irq_domain_add()
     ...  call include/linux/irqdomain.h:74:irq_domain_to_irq()
     ...  ERROR 77:  if (d->ops->to_irq)

links:
     *  [I2C](http://cateee.net/lkddb/web-lkddb/I2C.html)
     *  [TWL4030](http://www.ti.com/general/docs/marketurl.tsp?name=twl4030)
     *  [IRQ domain](http://lxr.gwbnsh.net.../IRQ-domain.txt)
```

Figure 3.4: VBDb record for bug 6252547b8a7.

3.4 Dimensions of analysis

I begin by selecting a number of properties of bugs to understand, analyze and document in bug reports. These are described below and exemplified by data from the VBDb database. Figure 3.4 shows an example record for a NULL-pointer dereference bug found in a Linux driver, which was traced back to errors both in the feature model and the mapping. Figure 3.5 shows the simplified version of this bug.

Type of bug. In order to understand the diversity of bugs I establish the type of each bug (field *type*) according to the *Common Weakness Enumeration* (CWE)[8]—a taxonomy of numbered software weaknesses and vulnerabilities. However, since CWE is mainly concerned with security, I had

[8] http://cwe.mitre.org/

to add a few additional types of bugs, including type errors, among others. The types of bugs that I found are listed in Fig. 3.1—my additions lack an identifier in the CWE column.

Bug description. Understanding a bug requires rephrasing its nature in general software engineering terms (field *descr*), so that the bug becomes understandable for non-experts. I add a one-line header to the description, to help identification and listing of bugs. I obtain such a description by studying the bug in depth, and following additional available resources (such as mailing list discussions, available books, commit messages, documentation and online articles). Whenever use of domain-specific terminology is unavoidable, I provide links to the necessary background. Obtaining the description is often non-trivial.

Presence condition. I investigate under what presence condition the bug appears (field *config*). This enables further investigation of variability properties of the bug, for example, the number of features and nature of dependencies that enable the bug. Obtaining the presence condition of a bug requires examining the source code and the Makefiles; as well as the Kconfig files to check for feature dependencies. For instance, Linux commit 6252547b8a7a (cf. Figure 3.4) fixes a 2-degree bug that occurs when enabling TWL4030_CORE, and disabling OF_IRQ.

Bug-fix layer(s). I analyze the bug-fixing commit to establish whether the source of the bug is in the code, in the feature model, or in the mapping (field *layer*). The bug of Fig. 3.4 has been fixed both in the *model* and in the *mapping*. The commit message asserts that: first, TWL4030_CORE should not depend on IRQ_DOMAIN (fixed in the model); and, second, that the assignment of the variable `ops` to `&irq_domain_simple_ops` is part of the IRQ_DOMAIN code and not of OF_IRQ (fixed in the mapping).

Bug trace. I analyze and document the execution trace that leads to the bug (field *trace*). Reconstructing the trace is key in understanding the nature and complexity of the bug; and, once documented, it allows other researchers to understand the bug much faster.

There are two types of entries in bug traces: function calls and statements. Function call entries can be either static (tagged *call*), or dynamic (*dyn-call*) if the function is called via a function pointer. A statement entry highlights relevant changes in the program state. Every entry starts

```
1  #define NULL (void*)0
2
3  #ifdef CONFIG_TWL4030_CORE
4  #define CONFIG_IRQ_DOMAIN
5  #endif
6
7  #ifdef CONFIG_IRQ_DOMAIN // ENABLED
8  int irq_domain_simple_ops = 1;
9
10 void irq_domain_add(int *ops)
11 {
12         int irq = *ops; // (4) ERROR
13 }
14 #endif
15
16 #ifdef CONFIG_TWL4030_CORE // ENABLED
17 int twl_probe()
18 {
19         int *ops = NULL; // (2)
20
21 #ifdef CONFIG_OF_IRQ // DISABLED
22         ops = &irq_domain_simple_ops;
23 #endif
24
25         irq_domain_add(ops); // (3)
26 }
27 #endif
28
29 int main()
30 {
31 #ifdef CONFIG_TWL4030_CORE // ENABLED
32         twl_probe(); // (1)
33 #endif
34         return 0;
35 }
```

Figure 3.5: Simplified version for bug 6252547b8a7.

with a non-empty sequence of dots indicating the nesting of function calls, followed by the location of the function definition (file and line) or statement (only the line). The statement in which the bug is manifested is marked with an *ERROR* label.

Simplified bug. I synthesize a simplified version of the bug, a small C program, that exhibits the same essential behavior, and the same essential problem. Simplified bugs are self-contained C files that do not depend on Linux kernel code, nor any other library than libc. The entire set of simplified bugs constitute an easily accessible benchmark suite derived from real bugs, which can be used to evaluate proof-of-concept tools on a smaller scale.

Simplified bugs are derived systematically from the bug trace. Along this trace, I preserve relevant statements and control-flow constructs, CPP conditionals, feature mapping information, and function calls. I keep the original identifiers for features, functions and variables. However, I abstract away complex code patterns (e.g. dynamic dispatching via function pointers) whenever this is not essential to capture the bug. For this reason, if a tool finds one of these simplified bugs, that does not imply that it will find the real bug too. When there exist dependencies between features, these are encoded operationally, using `#define` directives (cf. lines 4–6 of Figure 3.5). This makes the simplified bugs independent of any particular configuration modeling notation, such as Kconfig.

Traceability. The record includes the URL of the repository in which the bug fix is applied (field *repo*), the commit id (field *hash*), and links to relevant context information about the bug (field *links*), in order to support independent verification of my analysis.

3.5 Diversity of bugs in VBDb

I start by characterizing the diversity of VBDb bugs from three different perspectives: the types of errors, their location in the Linux source tree, and the configuration options involved.

> *Observation 1:*
> VBDb bugs are not limited to any particular type of errors.

Table 3.1 lists the type of bugs I have recorded, along with occurrence frequencies in the collection. The 43 bugs fall within 21 different error types, in seven wider bug classes. For example, 11 bugs have been classified under the category of *memory errors*, four of which are `NULL` pointer dereferences. Note that VBDb bugs cover a wide range of qualitatively different types of bugs.

Note that at least 11 of these bugs (*declaration* and *type* errors) are caught by the C compiler, and some are fatal (e.g. *undefined function*). The buggy code passed several code reviews and made it into the official Linux repositories, presumably because neither the author nor the reviewers compiled the kernel in the right configuration.

Observation 2:
VBDb bugs are not confined to any specific location (file or subsystem) in the Linux kernel.

Figure 3.6 shows in which subsystems the bugs are located and the relative size of each subsystem as of March 2014—I approximate subsystems by directories. The size of each subsystem is measured in lines of code (LOC).[9] For example, with six squares, the `kernel/` subsystem has approximately 150 KLOC and represents about 1% of the Linux code. Superimposed onto the size visualization, the figure also shows in which directories the bugs occur. With five red (dark) squares, the directory `kernel/` thus houses five of the bugs in VBDb.

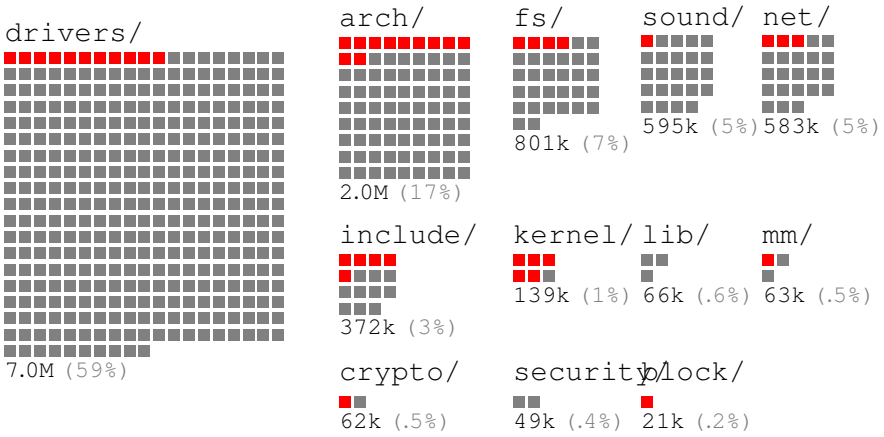
VBDb bugs occur in ten of the main Linux subsystems. Note that Linux subsystems are often maintained and developed by different people, which adds to the diversity of VBDb. I have found no bug in nine directories, but those represent less than the 3% of the Linux kernel code in total. (Of course, there are bugs fixed in those directories, but likely I have not found any because of the small sample size and that the search process was largely randomized in terms of bug location.) Further, three of

[9] As reported by cloc (`http://cloc.sourceforge.net/`), version 1.53.

Table 3.1: Types of bugs among the 43 bugs. The first column gives the frequency of these bugs in the VBDb collection.

11	**memory errors:**	**CWE**
4	null pointer dereference	476
3	buffer overflow	120
3	read out of bounds	125
1	write on read only	–
10	**resource mgmt. errors:**	**CWE**
5	uninitialized variable	457
1	memory leak	401
1	use after free	416
2	duplicate operation	675
1	double lock	764
8	**logic errors:**	**CWE**
5	fatal assertion violation	617
2	weak assertion violation	617
1	behavioral violation	440

7	**declaration errors:**	**CWE**
4	undefined function	–
2	undeclared identifier	–
1	mult. function definitions	–
4	**type errors:**	**CWE**
2	incompatible types	843
1	wrong func. arg. number	685
1	void pointer dereference	–
2	**dead code:**	**CWE**
1	unused function	561
1	unused variable	563
1	**arithmetic errors:**	**CWE**
1	numeric truncation	197



Smaller:
- `virt/` (6.8k), `ipc/` (6.4k), `init/` (2.0k), and `usr/` (0.6k).

Infrastructure:
- `tools/` (102k), `scripts/` (44k), and `samples/` (2.1k).

Figure 3.6: Location of the 43 bugs in the main Linux directories as of March 2014. Each square represents 25 thousand lines of code. The precise number of LOC and its percentage of the total is given below the squares. A red (dark) square symbolizes the occurrence of one of the bugs.

them (`tools/`, `scripts/`, and `samples/`) contain example and support code (build infrastructure, diagnostic tools, etc.) that does not run on a compiled kernel.

> *Observation 3:*
> VBDb bugs are not restricted to a few error prone features.

Table 3.2 shows the complete list of configuration options involved in the bugs. VBDb bugs involve a total of 78 qualitatively different features, ranging from *debugging* options (e.g. QUOTA_DEBUG), to *device drivers* (e.g. TWL4030_CORE), to *network protocols* (e.g. VLAN_8021Q), to *computer architectures* (e.g. PARISC). Three features are involved in three of the bugs, nine features occur in two bugs, and the remaining 66 are involved in only a single bug.

While the VBDb bug collection is not meant to be representative of the entire population of Linux bugs (cf. Sect. 3.2.3), it does exhibit high diversity. These 43 bugs involve 21 types of errors, 10 Linux subsystems, and 78 configuration options. This diversity supports the subsequent

qualitative investigation of the three research questions, in sections 3.6, 3.7, and 3.8.

3.6 RQ1: Variability characteristics of bugs in Linux

I have arrived at six observations (4–9) regarding the variability properties of VBDb bugs, and by extension Linux bugs, that make my answer to research question RQ1. Overall, the main conclusion is that reasoning about variability in Linux requires more than local examination of a piece of code (observations 4–7). Feature implementations are intermixed and span multiple subsystems, and reasoning about feature interactions requires looking not only at the code, but also at the Kconfig feature model, and the Makefiles. A secondary, but interesting, outcome of this study was the realization that disabling certain features may be an effective way of triggering bugs in Linux (observations 8–9). I believe that taking these observations into account will benefit the design of future variability-aware tools. (While this has not been the objective of my PhD project, it is one of the main objectives of project VARIETE, cf. Section 1.1.)

Observation 4:
The implementation of features in Linux is scattered across

Table 3.2: Configuration options involved in the bugs.

64BIT	IP_SCTP	S390
ACPI_VIDEO	JFFS2_FS_WBUF_VERIFY	S390_PRNG
ACPI_WMI	KGDB	SCTP_DBG_MSG
AMIGA_Z2RAM	KPROBES	SECURITY
ANDROID	KTIME_SCALAR	SHMEM
ARCH_OMAP2420	LBDAF	SLAB
ARCH_OPAM3	LOCKDEP	SLOB
ARM_LPAE	MACH_OMAP_H4	SMP
BACKLIGHT_CLASS_DEVICE	MODULE_UNLOAD	SND_FSI_AK4642
BCM47XX	NETPOLL	SND_FSI_DA7210
BDI_SWITCH	NUMA	SSB_DRIVER_EXTIF
BF60x	OF	STUB_POULSBO
BLK_CGROUP	OF_IRQ	SYSFS
CRYPTO_BLKCIPHER	PARISC	TCP_MD5SIG
CRYPTO_TEST	PCI	TMPFS
DEVPTS_MULTIPLE_INSTANCES	PM	TRACE_IRQFLAGS
DISCONTIGMEM	PPC64	TRACING
DRM_I915	PPC_256K_PAGES	TREE_RCU
EP93XX_ETH	PREEMPT	TWL4030_CORE
EXTCON	PROC_PAGE_MONITOR	UNIX98_PTYS
FORCE_MAX_ZONEORDER=11	PROVE_LOCKING	VLAN_8021Q
HIGHMEM	QUOTA_DEBUG	VORTEX
HOTPLUG	RCU_CPU_STALL_INFO	X86
I2C	RCU_FAST_NO_HZ	X86_32
IOSCHED_CFQ	REGULATOR_MAX8660	XMON
IPV6	REISERFS_FS_SECURITY	ZONE_DMA

> subsystems, therefore bugs may involve *non-locally defined features* (i.e., features defined in another subsystem than where the bug occurred).

A total of 30 (70%) of VBDb bugs involve non-locally defined features. This means that there is often functionality and features involved from different subsystems, than the one where the bug manifests. Identifying such bugs requires cross-subsystem knowledge, while it seems that most Linux developers are dedicated to a single subsystem.

For example, bug 6252547b8a7 (Fig. 3.5) occurs in the `drivers/` subsystem, but one of the interacting features, IRQ_DOMAIN, is defined in `kernel/`. Another example is bug 0dc77b6dabe, which manifests when loading the *extcon-class* module (`drivers/`), and is caused by an improper use of the *sysfs* virtual filesystem API—feature SYSFS in `fs/`.

> *Observation 5:*
> Linux coding conventions encourage the use of configuration-dependent definitions, introducing a form of variability that is implicit and cannot be observed by the naked eye.

The following coding guideline on `#ifdef` usage from *How to Get Your Change Into the Linux Kernel*[10] advises:

> *"Code cluttered with ifdefs is difficult to read and maintain. Don't do it. Instead, put your ifdefs in a header, and conditionally define 'static inline' functions, or macros, which are used in the code."*

This and other Linux guidelines encourage the introduction of *configuration-dependent definitions* (functions, macros, types, and so on). At the usage site, there is no indication of the configuration-dependent nature of these code entities—not even a name convention is used. Developers may reason about the code assuming the *most common* definitions for configuration-dependent entities, and ignoring the alternative definitions in *exceptional* configurations.

A common source of variability build errors is the invocation of functions from configurations in which are not defined. For instance, in bug 242f1a34377, function `crypto_alloc_ablkcipher()`, which is specific to feature CRYPTO_BLKCIPHER, is called by a piece a code (in another file) that assumes that this function always exist, leading to a build error when CRYPTO_BLKCIPHER is *disabled*.

[10] `https://www.kernel.org/doc/Documentation/SubmittingPatches`

39	**single layer:**
28	code
5	mapping
6	model
4	**multiple layers:**
2	code & mapping
1	mapping & model
1	code & mapping & model

Figure 3.7: Bug-fixing layers for VBDb bugs.

Another example is bug 0f8f8094d28, where an array is iterated by a `for` loop using an upper-bound that is not the right one for all possible configurations. In PowerPC architectures, and only for a particular virtual page size, the `for` loop will access the array out of its bounds.

> *Observation 6:*
> Linux bugs are fixed *not* only in the *code*; some are fixed in the *mapping*, some are fixed in the *model*, and some are even fixed in a *combination* of these layers.

Figure 3.7 shows whether the bugs in VBDb were fixed in the *code*, *mapping*, *model*, or combinations thereof (cf. Section 3.1). Remarkably, 15 bugs (35%) in VBDb involved fixes in the mapping or in the model, four of which required fixes in multiple layers.

For instance, Linux bug-fix 6252547b8a7 removes a feature dependency (TWL4030_CORE no longer depends on IRQ_DOMAIN) and changes the mapping to initialize the structure field `ops` when IRQ_DOMAIN (rather than OF_IRQ) is enabled. Linux commit 63878acfafb removes the mapping of some initialization code to feature PM (power management), and adds a function stub. Bug e68bb91baa0 was fixed in all the three layers!

The stratification into *code*, *mapping*, and *model* may obscure the cause of bugs, because an adequate analysis of a bug requires understanding all three layers. [MBW16] This complexity may cause a developer to fix a bug in the wrong place. For instance, the dependency of TWL4030_CORE on IRQ_DOMAIN removed by bug-fix 6252547b8a7 was added by commit aeb5032b3f8. Apparently, aeb5032b3f8 introduced this dependency into the Kconfig model to fix a build error, but this had undesirable side-effects.

8	**single-feature bugs:**
8	1-degree
35	**feature-interaction bugs:**
22	2-degree
9	3-degree
1	4-degree
3	5-degree

Figure 3.8: Variability degrees of VBDb bugs.

> *Observation 7:*
> The large number of features in Linux, together with the scattering of feature implementations across subsystems, make it prone to feature interaction bugs.

Figure 3.8 summarizes the degrees of bugs in VBDb. As many as 35 bugs (81%) are caused by feature interactions, and 13 (30%) involve three features or more. Feature-interaction bugs are inherently more complex to find and reason about [MBW16]. The number of potential interactions to consider is *exponential* in the number of features involved. This impacts both developers and analyzers that, consequently, have to cope with this combinatorial blow up.

For instance, Linux bug 6252547b8a7 (cf. Fig. 3.5) is a feature interaction bug. The code slice containing the bug involves three different features, and represents four variants (corrected for the feature model), but only one of the variants presents a bug. The `ops` pointer is dereferenced in variants with TWL4030_CORE enabled, but it is not properly initialized unless OF_IRQ is enabled.

Another example is bug ae249b5fa27, a 3-degree bug where an assertion is violated by the interaction of DISCONTIGMEM (efficient handling of discontinuous physical memory), and the ability to monitor memory utilization through the `proc/` virtual filesystem (feature PROC_PAGE_MONITOR), in PA-RISC architectures (feature PARISC).

> *Observation 8:*
> When coding, it is easy to assume that certain functionality is always present, thus disabling common features from the kernel is likely to uncover variability bugs.

Figure 3.9 lists and groups the structure of the presence conditions of VBDb bugs. A total of 22 bugs (51%) are triggered by disabling one or

21	**some enabled:**
5	a
10	$a \wedge b$
5	$a \wedge b \wedge c$
1	$a \wedge b \wedge c \wedge d \wedge e$
20	**some-enabled-one-disabled:**
3	$\neg a$
13	$a \wedge \neg b$
3	$a \wedge b \wedge \neg c$
1	$a \wedge b \wedge c \wedge d \wedge \neg e$
2	**other configurations:**
1	$\neg a \wedge \neg b$
1	$a \wedge \neg b \wedge \neg c \wedge \neg d \wedge \neg e$

Figure 3.9: Presence conditions under which VBDb bugs occur.

more features, 20 of which require disabling just one feature. The other 21 bugs are triggered by enabling one or more features, five of which (12% of the total) are just errors in the implementation of a feature. Note that a presence conditions of the form $(a \vee a') \wedge \neg b$ are classified as $a \wedge \neg b$. Similarly, presence conditions of the form $(a \vee a') \wedge b$. (For this reason, Fig. 3.8 and Fig. 3.9 may appear inconsistent.)

For instance, in bug 6252547b8a7 (Fig. 3.5), disabling OF_IRQ causes a NULL pointer dereference, because the initialization of a pointer is mapped to this feature. Another example is bug 60e233a5660, where disabling HOTPLUG leads to a buffer overflow. Interestingly, the code associated with HOTPLUG is unconditionally present in recent versions of the Linux kernel.

> *Observation 9:*
> A test sampling strategy based on disabling isolated groups of features could be effective at uncovering variability bugs.

Let us consider a *one-disabled* test sampling strategy, where we test configurations in which *at least one* feature is disabled (and preferably exactly one, if the feature model permits it). The size of such test sample is bounded by the number of features, and would cover 96% of bugs in VBDb. Testing of highly-configurable systems is often approached by testing *maximal configurations*, in which as many features as possible are

enabled—i.e., *all-enabled*, in an attempt to maximize code coverage. Maximal configuration testing covers just 21 bugs (49%) in VBDb (cf. Fig. 3.9).

In Linux, `make allyesconfig` follows a simple algorithm that produces one pseudo-maximal configuration. Yet, the enumeration of all *one-disabled* configurations (e.g., using on a MaxSAT solver) could result in tens of thousands of configurations to test (as many configurations as features, in the worst case). In practice, this could be implemented similarly to `allyesconfig`, by generating a handful of configurations where pre-defined (groups of) features are disabled (e.g., 64BIT, SMP, PCI, IPV6, etc.).

Remarkably, Medeiros et al. [MKR+16] executed a comparative quantitative study of the effectiveness of various test sampling strategies for configurable systems, including the above *one-disabled* strategy—known to them through our ASE 2014 paper [ABW14]. According to their study, *one-disabled* is the only non-trivial method that is able to scale to all of the Linux kernel, and was more effective at finding bugs than both *all-enabled* and random sampling.

3.7 RQ2: Challenges in analyzing Linux source code

I also analyzed the VBDb bug collection from a programming-language perspective. This yielded six major observations (10–15) that describe non-trivial challenges in analyzing Linux code. These challenges are largely part of the folklore knowledge in the static analysis community but, some of them, are vaguely described in the literature. This study shows how these challenges arise in Linux due to specific coding patterns, using excerpts of real bugs from VBDb.

> *Observation 10:*
> A static code analyzer needs to parse CPP conditional directives (i.e. `#if` and so on) in order to recognize configuration-dependent code.

In Fig. 3.5, for instance, it is necessary to associate the initialization of the `ops` pointer with feature OF_IRQ. Parsing *unprocessed* or *partially* preprocessed C files, that maintain some preprocessor directives, is a difficult but known problem. Sound approaches that employ fork-merge parsers are, unfortunately, too slow [KGR+11, GG12]. The alternative is heuristic and fault-tolerant parsing [PWC91, Pad09]. Yet, to my knowledge,

```
1  void inet_ehash_locks_free() {
2          #ifdef CONFIG_NUMA      // ENABLED
3          if (size > PAGE_SIZE)
4                  vfree(hashinfo->ehash_locks);
5          else
6          #else   // DISABLED
7                  kfree(hashinfo->ehash_locks);
8          #endif
9          hashinfo->ehash_locks = NULL;   // LEAK
10 }
```

Figure 3.10: Bug 218ad12f42e: when NUMA is enabled, if the condition in line 3 evaluates to *false*, line 10 reassigns `hashinfo->ehash_locks` thus leaking memory.

there is no production-ready compiler front-end available that supports CPP-aware parsing.

Note that writing such a parser is difficult for multiple reasons [KGR+11]. The first challenge is to partially preprocess files, including headers and expanding macros, while maintaining CPP conditionals. (Note that CPP conditionals can surround `#include`s and `#define`s!) The second challenge is parsing *unstructured* (or *undisciplined*) uses of `#if` conditionals, that do not wrap complete C syntactic entities. (It is relatively simple to extend the C grammar to parse structured uses of `#if` conditionals.) For instance, Figure 3.10 shows the use of unstructured CPP involved in bug 218ad12f42e. In this case, the `#ifdef` block wraps only part of the `if` statement.

> *Observation 11:*
> A static code analyzer needs to be *configuration-sensitive* in order to identify how variability impacts the data-flow and the control-flow.

The VBDb bug collection evidences how configuration options can affect the program state and control-flow, with unexpected consequences. This is particularly the case for VBDb bug examples of use before initialization, `NULL` pointer dereference, and buffer overflow. For instance, in Fig. 3.5, the value of `ops` in line 25 (hence, in line 12) may or not be `NULL` depending on OF_IRQ. Figure 3.11 shows a bug in VBDb, where feature HOTPLUG determines which function is called in line 17. The function that is called when HOTPLUG is disabled causes a buffer overflow in line 20.

There are two ways of dealing with variability at this level. The simplest is a technique known as *configuration lifting* [PS08], and that

```
1  #if defined(CONFIG_HOTPLUG)     // DISABLED
2  int add_uevent_var(struct kobj_uevent_env *env, const char *str)
3  {
4          int len = sprintf(&env->buf[env->buflen], str);
5          env->buflen += len + 1;
6          return 0;
7  }
8  #else   // ENABLED
9  int add_uevent_var(struct kobj_uevent_env *env, const char *str)
10 {
11         return 0;
12 }
13 #endif
14
15 void input_add_uevent_modalias_var(struct kobj_uevent_env *env)
16 {
17         int len;
18
19         if (add_uevent_var(env, "MODALIAS="))
20                 return -ENOMEM;
21
22         len = sprintf(&env->buf[env->buflen-1], "X"); // OVERFLOW
23         env->buflen += len;
24 }
25
26 void show_uevent()
27 {
28         struct kobj_uevent_env *env;
29         env = kzalloc(sizeof(struct kobj_uevent_env), GFP_KERNEL);
30         input_add_uevent_modalias_var(env);
31 }
```

Figure 3.11: Bug 60e233a5660: In line 26, function `show_uevent` allocates and zero-initializes a `kobj_uevent_env` structure, `env`. This structure maintains a string buffer `env->buf` of length `env->buflen`. There are two versions of function `add_uevent_var` depending on feature HOTPLUG. If HOTPLUG is disabled `add_uevent_var` does nothing, `env->buflen` maintains its zero value, and consequently `input_add_uevent_modalias_var` will write at `env->buf[-1]` in line 20, outside of the buffer bounds.

consists in analyzing a meta-program that subsumes all program configurations. Essentially, this meta-program encodes features as Boolean program variables, and CPP conditionals as regular `if` statements in C [IMD+17]. The second option, significantly more laborious, consists in adapting static analyzers to cope with variability [BRT+13, BTR+13]. This adaptation typically requires that the many data structures of the analyzer (e.g., the symbol table) are indexed by configurations.

```
1  void pts_sb_from_inode(struct inode *inode)
2  {
3          #ifdef CONFIG_DEVPTS_MULTIPLE_INSTANCES
4          if (inode->i_sb->s_magic == ···) // NULL DEREFERENCE
5                  ···
6          #endif
7  }
8
9  void pty_close(struct tty_struct *tty)
10 {
11         #ifdef CONFIG_UNIX98_PTYS
12         pts_sb_from_inode(tty->driver_data);
13         #endif
14 }
15
16 void tty_release(struct tty_struct *tty)
17 {
18         tty->ops->close(tty); // CALLS TO pty_close
19 }
20
21 #ifdef CONFIG_UNIX98_PTYS
22 void ptmx_open(struct inode *inode)
23 {
24         struct tty_struct *tty;
25         tty = tty_init_dev(ptm_driver);
26
27         if (*)
28                 goto err_release;       // TAKEN
29
30         tty->driver_data = inode;       // NOT EXECUTED
31         return;
32
33         err_release:
34                 tty_release(tty);
35 }
36 #endif
```

Figure 3.12: Bug 7acf6cd80b2: In line 25, function `tty_init-dev` allocates and *partially* initializes a `tty_struct` structure, `tty`: `tty->ops` is set to `ptm_drivers->ops`, while `tty->driver_data` is set to `NULL`. Upon some error condition the `tty` is released, in line 34. Invocation of function pointer `tty->ops->close` dynamically calls `pty_close`, where `tty->driver_data` is passed to function `pts_sb_from_inode`. At this point, `tty->driver_data` is still a pointer to `NULL`. If DEVPTS_MULTIPLE_INSTANCES is enabled this `NULL` pointer is dereferenced in line 4 causing a kernel crash.

Observation 12:
A static code analyzer needs to resolve the potential targets of function pointers in a context-sensitive manner, in order to follow dynamic calls to functions.

Ten bugs in VBDb (33% of runtime bugs) involve dynamic calls through function pointers. Bug 7acf6cd80b2 (cf. Figure 3.12) is one of these ten cases. In this example, `tty_struct` is an abstraction for manipulating arbitrary terminal devices (*TTY*), which can have different implementations. In the context of `ptmx_open`, the `tty` object represents a pseudo-terminal emulator, and `tty->ops->close` points to function `pty_close`. Building a precise call graph for Linux is non-trivial and requires both a configuration- and context-sensitive analysis of function calls.

This use of function pointers arises mainly from a common idiom in C, and particularly in Linux, that consists in using *function pointer tables* as generic interfaces for manipulating different resources uniformly. Typically, pointers to data and *operations* (i.e., functions) are aggregated in structure types, in order to obtain data and logic encapsulation. For instance, in Fig. 3.12 the `tty_struct` structure holds a pointer to driver-specific data (field `driver_data`, a void pointer), and a pointer to a *method dispatching table* (field `ops`, a pointer to a `tty_operations` structure). In addition to the aforementioned *TTY* abstraction, another classic example is the abstraction of *file descriptor*. Basically, any device driver in Linux exposes its functionality through an interface of this kind.

Function pointer tables are also used to implement generic traversals over kernel data structures, akin to the *visitor* pattern. For instance, bug 8c8296223f3 is a buffer overflow during the execution of an `mm_walk` visitor. An `mm_walk` object holds a sets of callbacks, which `walk_page_range` executes on the different levels of the virtual memory hierarchy.

> *Observation 13:*
> A static code analyzer needs to understand the use of type casts to workaround the lack of generics and subtyping in C, in order to track the flow of data effectively.

Generic data manipulation in C relies on the use of type casts to remove, and recover, type information associated with memory addresses. For instance, Figure 3.13 shows a fragment of bug 657e9649e74, where a `long` integer is decoded into a function pointer that is subsequently invoked (lines 10–11). This bug occurs in the internal timer mechanism of the Linux kernel. Because timers manipulate different types of data, a `timer_list` holds pointers to arbitrary data, that are encoded as `long` integers. For this reason, pointer analysis of C code largely ignores the

```
1  void tcp_twsk_destructor()
2  {
3          #ifdef CONFIG_TCP_MD5SIG        // ENABLED
4          put_cpu(); // DECREASES preempt_count()
5          #endif
6  }
7
8  void inet_twdr_hangman(long data)
9  {
10         void (*fn)();
11         fn = (void(*)())data;
12         fn();   // CALLS TO tcp_twsk_destructor
13 }
14
15 void __run_timers(struct list_head *head)
16 {
17         struct timer_list *timer;
18         list_for_each_entry(timer, head, entry) {
19                 int pc = preempt_count();
20                 timer->function(timer->data);
21                 if (pc != preempt_count())
22                         BUG();  // KERNEL PANIC
23         }
24 }
```

Figure 3.13: Bug 657e9649e74: Function `__run_timers` iterates through a list of `timer_list` entries and executes each callback function (`timer->function`). Each timer has an associated piece of data (`timer->data`) that is passed to the callback. One of these callbacks is `inet_twdr_hangman`, which obtains and invokes a function pointer to `tcp_twsk_destructor`, that is stored in the timer's data. If feature TCP_MD5SIG is enabled, `tcp_twsk_destructor` will decrement the *preemption counter*, violating a code invariant (lines 19, 21–22).

types of variables and expressions, and instead tracks their memory *shape* [Ste96b].

Similarly, type casts between structure types are used to achieve structural subtyping in C. While C forbids direct casts between structure types, it is legal to cast between pointers to arbitrary structure types. This is particularly subtle because, in general, the correspondence between the fields of two structures is implementation dependent [Ste96a, YHR99]. For instance, in Linux, generic network sockets are represented by the `sock` structure (the *base type* of sockets), but there are also specific (sub)types of sockets that have their own representations. Specialized sockets (e.g. `struct inet_timewait_sock`) can be passed to generic socket functions by explicitly *"up-casting"* them (to `struct sock`). For this to work, the first field of any socket structure

holds a `sock_common` structure, which contains common information to all socket types.

> *Observation 14:*
> A static code analyzer needs to model structure objects precisely, in order to find non-trivial bugs in Linux code.

It is commonplace, in virtually any programming language, to use a hierarchy of *record* types (i.e., C structures) to represent and relate domain concepts. Actually, all of the bugs discussed in this section do involve multiple accesses to structure fields. (Also bug 6252547b8a7 does involve structures, even though the simplified version shown in Fig. 3.5 has abstracted them away.) Pointer analyses that do not model structure objects precisely do not work well in practice—they are too imprecise. However, doing sound pointer analysis of C structure objects is considerably more difficult than the simplistic approach of *collapsing* their fields [Ste96a, YHR99].

Ideally, a static analyzer should also model recursive and dynamic data structures precisely. Linux code makes extensive use of them. For instance, in bug 657e9649e74 (cf. Fig. 3.13), `__run_timers` receives a linked list of `timer_struct` objects to run, one of which will call to `inet_twdr_hangman`. (This code uses the Kernel generic circular double-linked list implementation, see `include/linux/list.h`.) Reasoning about such data structures requires *(deep-)shape analyses* [CDOY09, CDOY11]. Fortunately, as this PhD project shows, it is possible to find non-trivial bugs in Linux without having a precise model of linked data structures—as we will see in Chapter 7.

> *Observation 15:*
> A static code analyzer needs to perform inter-procedural analysis, in order to find non-trivial bugs in Linux code.

Figure 3.14 shows a classification of VBDb runtime bugs according to the depth of their error trace. A total of 25 bugs (83%) of the 30 runtime bugs in VBDb have an error trace that involves, at least, one nested function call. Note that all of the bugs discussed in this section cross the boundaries of a single function. For instance, bug 657e9649e74 involves seven nested function calls—five more than shown in Fig. 3.13, two of which are dynamic calls through function pointers. There exist many interprocedural analysis techniques, but these are mainly whole-program analyses that would not run fast enough on a large Linux-size codebase.

5	**single-function bugs:**
5	0-call deep
25	**cross-function bugs:**
6	1-call deep
5	2-call deep
4	3-call deep
5	5-call deep
1	6-call deep
1	7-call deep
1	9-call deep
2	10-call deep

Figure 3.14: Function call depths of VBDb runtime bugs.

Note also the need for inter-file analysis. Bug traces often touch functions defined in multiple C files and subsystems; like bug 657e9649e74, which concerns functions from three files and two subsystems (`kernel/` and `net/`). This definitely calls for static analyzers that can work compositionally.

3.8 RQ3: Opportunities for bug finders in Linux

The conclusions of studying RQ2 (Sect. 3.7) may be discouraging at first, but do offer some insight into the opportunities for bug finding technology. It was the study of this third RQ that lead to the bug finding technique depicted in Sect. 1.4, and that constitutes the core of this thesis. Code scanners (cf. Sect. 2.3.2) must run fast and cannot afford the cost of reasoning about complex data dependencies. Thus, the study of RQ3 stems from the following observation:

> *Observation 16:*
> Many bugs in Linux can be traced down to API misuses, where often there are no complex data flows involved. These bugs are difficult to intercept simply because the sequences of operations that trigger them span multiple functions.

About half of VBDb runtime bugs are violations of control-dominated API rules. (Also known as *finite-state* or *typestate* properties [SY86].) In the following, I may refer to these type of programming errors as *resource*

manipulation bugs (CWE-399).[11] Resource safety has been the target of many automated verification techniques [SY86, ECCH00, BR01b, FTA02, XA05] with relative success (see Section 2.3). Software model checkers [BR01b, XA05] have yielded the best results so far but are slow in comparison with code scanners. From a pragmatic perspective, these tools put too much effort in tracking changes to program data, whereas finding resource manipulation bugs is more about tracking sequences of operations.

> *Observation 17:*
> Let us assume that there exist a program abstraction that reveals the operations performed, directly or indirectly, by any function call. Then, violations of control-dominated API rules could be efficiently uncovered by intraprocedural matching of flow-based bug patterns.

Such an abstraction does exist, and it is based on the concept of *side-effect*. A side-effect (or just *effect*) denotes the execution of a certain operation that may change the program state. Side-effect analyses (cf. Sect. 2.2) infer the effects associated with every expression in a program; and allow to build an effect-based abstraction, where effects reveal the operations performed through function calls, like that of Fig. 1.3.

The remainder of this section describes a conceptual exercise of testing the effectiveness of this hypothetical bug-finding technique on VBDb bugs. The goal is to detect violations of API rules, or resource mismanipulation bugs, where a resource should be understood broadly. Table 3.3 shows which bugs from the VBDb collection could potentially be found by this technique. Bugs are grouped according to *weakness classes* from the CWE taxonomy. The following subsections discuss each bug class with examples, and provides the associated bug patterns, expressed in *Computational Tree Logic* (CTL).

3.8.1 Null-pointer dereference (CWE-476)

Three out of four `NULL`-pointer deference bugs in VBDb can be understood as resource manipulation bugs. Bug 76baeebf7df does not; it is a complex case for which a static analyzer would have to quite precisely track integer values, and the contents of arrays. Bug ee3f34e8572 is a

[11] `https://cwe.mitre.org/data/definitions/399.html`

single-function bug, where a pointer variable is set to NULL and later dereferenced in the same function. Bugs 6252547b8a7 and f7ab9b407b3 are similar to ee3f34e8572, but the pointer dereference happens inside several nested function calls. Figure 3.15 shows an excerpt of bug 6252547b8a7.

In order to find null-pointer deferences of this kind, we must start in a state in which some memory location x (a variable or any l-value expression) is set to NULL (operation $Null(x)$), and eventually dereferenced (operation $Use(\star x)$), without having been assigned in between (operation $Assign(x)$). The corresponding CTL query is:

$$\mathsf{EF}\ (Null(x) \wedge \mathsf{EX}\ (\neg Assign(x)\ \mathsf{EU}\ Use(\star x))) \tag{3.1}$$

3.8.2 Duplicate operation on resource (CWE-675)

There are three resource mis-manipulation bugs of this kind in VBDb. Bug d7e9711760a has been introduced in Fig. 1.2; and it is, more specifically,

Table 3.3: Resource manipulation bugs in VBDb that can be uncovered by the hypothetical bug-finding technique proposed in Obs. 17.

Bug class	**CWE**	**VBDb id**	**Sec**
Null-pointer dereference	476	76baeebf7df ee3f34e8572 6252547b8a7 f7ab9b407b3	3.8.1
Duplicate operation on resource	675	d7e9711760a 0dc77b6dabe 472a474c663	3.8.2
Resource leak	772	218ad12f42e	3.8.3
Use before initialization	908	e39363a9def 1c17e4d4437 30e053248da bc8cec0dff0	3.8.4
Other Linux-API misuses	—	208d89843b7 eb91f1d0a53	3.8.5

```
1  void irq_domain_to_irq(int *ops)
2  {
3          int irq = *ops; // Use(*ops)
4  }
5
6  void irq_domain_add(int *ops)
7  {
8          irq_domain_to_irq(ops); // Use(*ops)
9  }
10
11 void twl_probe()
12 {
13         int *ops = NULL;          // Null(ops)
14         irq_domain_add(ops);      // Use(*ops)
15 }
```

Figure 3.15: Bug 6252547b8a7: null pointer dereference.

a double-lock bug (CWE-764). Similarly, double-free bugs would also belong to this class. Bug 0dc77b6dabe is triggered by a load-unload-load sequence of a kernel module, where the *exit* function of the module does not cleanup correctly. Figure 3.16 shows an excerpt of the third case, bug 472a474c663. Function `enable_IR_x2apic` is part of an initialization sequence executed during boot, that should not be called more than once, otherwise resulting in a kernel panic.

Bugs in this category happen when some non-idempotent operation is applied on a resource ($Op(x)$), and later on applied again on the same resource, causing undefined behavior. In between the two applications the resource must not have been *reset* or its state altered ($Reset(x)$). The corresponding CTL bug pattern is:

$$\mathsf{EF}\ (Op(x) \wedge \mathsf{EX}\ (\neg Reset(x)\ \mathsf{EU}\ Op(x))) \tag{3.2}$$

For instance, for bug d7e9711760a, $Op(x) =$ `spin_lock`(x) and $Reset(x) =$ `spin_unlock`(x). For bug 472a474c663 (cf. Fig. 3.16), we have $Op() =$ `enable-IR-x2apic`$()$, and $Reset() = \bot$. To my knowledge, there is no reset operation in this case.

3.8.3 Resource leak (CWE-772)

Figure 3.17 shows an fragment of the one resource (memory) leak bug in VBDb, namely 218ad12f42e. Function `inet_ehash_locks_alloc` allo-

```
1  void enable_IR_x2apic(void) { /* ... */ }
2
3  void APIC_init_uniprocessor(void)
4  {
5          enable_IR_x2apic();      // Op
6  }
7
8  void smp_sanity_check(unsigned max_cpus)
9  {
10         if (!smp_found_config)
11                 APIC_init_uniprocessor();    // Op
12 }
13
14 void native_smp_prepare_cpus(unsigned int max_cpus)
15 {
16         enable_IR_x2apic();              // Op
17         smp_sanity_check(max_cpus);      // Op
18 }
```

Figure 3.16: Bug 472a474c663: duplicate initialization (kernel panic).

cates an array of locks (`hashinfo->ehash_locks`) in line 3, and function `inet_ehash_locks_free` clears the only reference to this array in line 9 without having freed it. This is clearly a resource mis-manipulation bug.

A resource leak of this kind happens when there is a path, starting from a point where some resource (e.g., a chunk of memory) is allocated at some memory location x ($Alloc(x)$), to a point where there are no live references to that resource, without having being freed ($\neg Free(x)$). The corresponding CTL bug pattern is:

$$\mathsf{EF}\ (Alloc(x) \wedge \mathsf{AG}\ \neg Free(x)) \tag{3.3}$$

3.8.4 Use before initialization (CWE-908)

There are four resource mis-manipulation bugs of this kind in VBDb. In bug e39363a9def, which affects a single function, there is an error path where variable `err` is not set to the appropriate error code. This causes function `netpoll_setup` to return an arbitrary value. Figure 3.18 shows an excerpt of 1c17e4d4437, where `print_cpu_stall_info` relies on a second function, `print_cpu_stall_fast_no_hz`, to initialize a

```
1  void inet_ehash_locks_alloc(struct inet_hashinfo *hashinfo)
2  {
3          hashinfo->ehash_locks = kmalloc(···, GFP_KERNEL);
4  }
5
6  void inet_ehash_locks_free(struct inet_hashinfo *hashinfo)
7  {
8          if (hashinfo->ehash_locks)
9                  hashinfo->ehash_locks = NULL;   // LEAK
10 }
11
12 void dccp(void)
13 {
14         struct inet_hashinfo dccp_hashinfo;
15         inet_ehash_locks_alloc(&dccp_hashinfo); // Alloc
16         inet_ehash_locks_free(&dccp_hashinfo);
17 }
```

Figure 3.17: Bug 218ad12f42e: memory leak.

string buffer in line 10. The buffer is never initialized but used in line 11. Bug 30e053248da is analogous to 1c17e4d4437, in this case it is function `reiserfs_security_init` that expected that function `security_old_inode_init_security` would initialize a structure field. Finally, bug bc8cec0dff0 causes a kernel panic when `jffs2_flash_write` attempts to access an unallocated buffer. Bug bc8cec0dff0 involves more than ten function calls, and is triggered by a non-obvious flow of control.

Use before initialization bugs happen when a resource is declared ($Decl(x)$) and later used $Use(x)$, without having been initialized $Init(x)$. The corresponding CTL bug pattern is:

$$\mathsf{EF}\ (Decl(x) \wedge \mathsf{EX}\ (\neg Init(x)\ \mathsf{EU}\ Use(x))) \tag{3.4}$$

For bug 1c17e4d4437 (cf. Fig. 3.18), operation $Decl(\texttt{fast\_no\_hz})$ corresponds to the declaration of the buffer in line 8, and $Use(\texttt{fast\_no\_hz})$ to the read of this buffer in line 11. The buffer could have been initialized (i.e. $Init(\texttt{fast\_no\_hz})$) by `sprintf`, for instance.

3.8.5 Other Linux-API misuses

This technique can also catch project-specific API misuse bugs. Two examples from VBDb are bugs 208d89843b7 and eb91f1d0a53, which are both

```
1  void print_cpu_stall_fast_no_hz(char *cp, int cpu)
2  {
3
4  }
5
6  void print_cpu_stall_info(int cpu)
7  {
8          char fast_no_hz[72];    // Decl
9
10         print_cpu_stall_fast_no_hz(fast_no_hz, cpu);
11         printk(KERN_ERR "\t%d: %s\n", cpu, fast_no_hz); // Use
12 }
```

Figure 3.18: Bug 1c17e4d4437: use before initialization.

violations of Linux interrupt management rules. Hardware interrupts (IRQs) should only be disabled during very short periods of time, typically, only during the execution of an *interrupt handler*. While IRQs are disabled, kernel code should not sleep, enter a busy wait, or perform long computations.

Figure 3.19 illustrates bug 208d89843b7, this one enables *software* interrupts (BHs) while IRQs are disabled. Hardware interrupts are disabled in line 18 by `spin_lock_irq`, and subsequently BHs are enabled in line 19, through a chain of function calls that leads to a call to `local_bh_enable()` in line 3. This could allow BH handlers to run with interrupts disabled. Software interrupts deal with time-consuming tasks and, therefore, must be interruptible. The corresponding CTL bug pattern is:

$$\mathsf{EF}\ (\mathit{IRQsOff} \wedge \mathsf{EX}\ (\neg \mathit{IRQsOn}\ \mathsf{EU}\ \mathit{BHsOn})) \tag{3.5}$$

Bug eb91f1d0a53 happens during kernel boot, when several initialization procedures are run with interrupts disabled. One of these procedures tries to allocate memory, and does it passing the `GFP_WAIT` flag to the memory allocator. Flag `GFP_WAIT` tells the allocator that, if no contiguous chunk of memory of the adequate size is available, it is allowed to sleep. The corresponding CTL bug pattern is:

$$\mathsf{EF}\ (\mathit{IRQsOff} \wedge \mathsf{EX}\ (\neg \mathit{IRQsOn}\ \mathsf{EU}\ \mathit{Sleep})) \tag{3.6}$$

```
1  void skb_checksum(struct sk_buff *skb)
2  {
3          local_bh_enable();      // BHsOn
4  }
5
6  void skb_checksum(struct sk_buff *skb)
7  {
8          kunmap_skb_frag(skb);   // BHsOn
9  }
10
11 void udp_checksum_complete(struct sk_buff *skb)
12 {
13         skb_checksum(skb);      // BHsOn
14 }
15
16 void udp_poll(struct sk_buff_head *rcvq, struct sk_buff *skb)
17 {
18         spin_lock_irq(&rcvq->lock);     // IRQsOff
19         udp_checksum_complete(skb);     // BHsOn
20         spin_unlock_irq(&rcvq->lock);   // IRQsOn
21 }
```

Figure 3.19: Bug 208d89843b7: BHs get enabled with IRQs disabled.

Chapter 4

A Shape and Effect System for C(IL)

Originally published in: VMCAI 2017[1]

At the core of EBA there is a new type-and-effect inference system for C in the style of Talpin and Jouvelot [TJ92, JT93]. Because of unsafe casts, the standard C type system provides only a meager description of run-time objects. Thus, as done in pointer analysis [Ste96a, Ste96b], types in EBA describe objects by their *shape* in memory—hence the name *shape*-and-effect system. Shapes are unaffected by type casts, being suitable for tracking aliasing relations. This system is *polymorphic* in *shapes*, *regions*, and *effects*; and it supports *sub-effecting*. Thanks to shape polymorphism it can handle common idioms, such as those used to workaround type genericity in C. Effect polymorphism and sub-effecting allow handling function pointers.

EBA analyzes programs in CIL (C Intermediate Language), an analysis-friendly subset of C [NMRW02]. CIL has a simpler syntax-directed type system than C, without implicit type conversions. Using CIL allowed me to scaffold a tool prototype faster, while still being able to handle the entire C (via a C-to-CIL front-end). The abstract syntax and type system of CIL are discussed in [NMRW02].[2] I will not discuss them here, but the chapter should be readable by any person who is familiar with the C language.

[1]Our paper entitled *"Effective Bug Finding in C Programs with Shape and Effect Abstractions"*[ABW17], presents a simplified version of this inference system, and is published in the 18th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2017).

[2]For a precise definition of the CIL abstract syntax one must look at the source code: `https://github.com/cil-project/cil/`, file `src/cil.mli`.

Regions	ρ	:	ρ
Effects	f_X	:	$\varepsilon(\overline{\rho})$
Effect sets	F	:	$\emptyset \mid \{f_X\} \mid \varphi \mid F_1 \cup F_2$
R-shapes	Z^R	:	$\bot \mid \mathsf{ptr}\ Z^L \mid \texttt{struct}\ A_{\mid n}\ \texttt{\{}\ \overline{Z_i^L\ x_i}\ \texttt{\}} \mid \zeta$
F-shapes	Z^F	:	$Z_1^L \times \cdots \times Z_n^L \xrightarrow{F} Z_0^R$
L-shapes	Z^L	:	$\mathsf{ref}_\rho\ Z^R \mid \mathsf{ref}_\rho\ Z^F$

Figure 4.1: The type language: regions, effects, and shapes.

4.1 The shape language

Types in this system are divided into three categories: *regions*, *effects*, and *shapes*. Regions describe memory locations, effects describe computations on regions, and shapes describe the representation of data and the computational effects of programs. Figure 4.1 shows their corresponding abstract syntax.

4.1.1 Regions

Consider the C program `int x; int *y = &x; S`, where S is an arbitrary statement. Within S's scope, both x and $*y$ denote the same memory cell—they *alias*. Due to aliasing, two different l-values may denote the same memory location at run-time. But, it would be infeasible to track the precise memory locations associated with each l-value, at each point in the program. Instead, this system tracks memory *regions* (ρ), which are abstract sets of possibly-aliased memory locations.

Shapes (cf. Sect. 4.1.3) are annotated with regions, and typing rules will impose aliasing constraints over these regions. Shapes are flow-insensitive and the constraints over regions are limited to equalities expressing aliasing relations, that are exploited and inferred by unification. Shape inference embeds an alias analysis that is similar to Steengaards's points-to analysis [Ste96b], but polymorphic.

4.1.2 Effects

Types describe *what* expressions compute, whereas effects describe *how* expressions compute [JG91]. From the type perspective, the expression `y = y + x` evaluates to an integer value. From the effect perspective, it reads from locations `x` and `y`, and writes to `y`. Effects are a framework to reason about such and similar aspects of computations. If variables `x` and `y` are stored in regions ρ_x and ρ_y, respectively; then EBA infers that `y = y + x` has effects $F = \{read_{\rho_x}, read_{\rho_y}, write_{\rho_y}\}$, which records reading variables `x` and `y`, and writing `y`. A set of effects is a *flow-insensitive abstraction* of an execution. It specifies the effects that *may* result from evaluating an expression (or statement), disregarding the flow of control.

We assume a finite number of *effect constructors* of finite arity, including nullary (Fig. 4.1). A constructor ε applied to a tuple[3] $\overline{\rho}$ of memory regions defines a discrete effect $\varepsilon_{\overline{\rho}}$ taking place on regions in $\overline{\rho}$. Built-in effects, inherent to the C language, include reading and writing of memory locations, $read_\rho$ and $write_\rho$; and calling to functions (e.g., through function pointers), denoted $call_\rho$. Other effects can be introduced to capture new kinds of bugs. The example of Fig. 1.3 used effects $lock_\rho$ and $unlock_\rho$ to represent lock manipulation actions. We use effect variables (φ, cf. Fig. 4.1) to stand for sets of effects, to achieve effect polymorphism. Effects are combined into sets (F), ordered by the usual set inclusion.

4.1.3 Shapes

A *shape* approximates the memory representation of an object [Ste96a, Ste96b]. Whereas, in the base C type system, an expression can be coerced to an arbitrary different type, in this system, the shape of an expression is (supposed to be) fixed and preserved across type casts (cf. Sect. 4.3). Shapes are split into r-value (Z^R), f-value (Z^F), and l-value (Z^L) shapes; defined in detail below. The shape language resembles the base C type language, without integer type but with shape variables (ζ).

R-value shapes

R-shapes (Fig. 4.1) denote the shape of r-value objects; i.e., C expressions that can be placed on the right-hand side of an assignment (or passed

[3] We use overline to denote tuples.

as arguments to a function). An *atomic* shape $\perp$ denotes objects that have no relevant structure, for instance integers, when these are not masquerading pointers to implement genericity (see below).

Pointer expressions have pointer shapes, ptr Z^L, where Z^L is the shape of the target reference cell of the pointer. A pointer represents an r-value of the *address* of a reference cell, which is in itself an l-value. Therefore, a pointer shape necessarily encloses a reference shape. Pointers may be cast to integers to emulate generics; such integer values will thus have a pointer shape.

There are no array shapes. In order to avoid depending on decision procedures for determining aliasing, this shape-and-effect system flattens array shapes and treats them as regular pointers. This makes the different elements of an array indistinguishable. Two l-values `a[i]` and `a[j]` will have the same shape in this system, independently of the value of `i` and `j`. This is sound, but introduces imprecisions that shall be addressed by the client of the analysis—the simplest way is through heuristics, cf. Chapter 6.

We use `struct` $A_{|n}$ `{` $\overline{Z_i^L\ x_i}$ `}` to denote the shape of an `struct` A object of which only the first n members are accessed. A struct shape associates an l-shape Z_i^L with each member x_i of the n-prefix of A. When the context allows, we may omit the n and write `struct` A `{` $\cdots$ `}` instead. (The reason why struct shapes may refer only to a prefix of the struct members will become clear in Sect. 4.3.) Each struct member is treated as a separate variable, and may be assigned a shape that is unrelated to the other members. This is unsound, [4] but it is essential to maintain both precision and simplicity. (Previous work maintains soundness at the cost of added complexity [Ste96a, YHR99].) In practice, a client of this analysis can keep track of the storage regions associated with a struct object, and recover soundness to the extent desired (e.g., soundness can be fully recovered by treating all of the members as aliases).

Shape variables ζ are used to make shapes polymorphic, they stand for arbitrary r-value shapes. For instance, functions manipulating a generic linked list are shape polymorphic, since they abstract from the shape of objects stored in the list.

[4] In reality, the members of a struct object are placed in a contiguous region in memory, and a pointer to one member can be used to update the others. For instance, `memset(&s.x,0,sizeof(struct A))` zero-initializes an object `s` of type `struct A`, given a pointer to the first member `x`.

F-value shapes

A function shape, $Z_1^L \times \cdots \times Z_n^L \xrightarrow{F} Z_0^R$, maps a tuple of reference shapes (i.e., $Z_1^L \times \cdots \times Z_n^L$), corresponding to the formal parameters, to a value shape (i.e., Z_0^R), corresponding to the result. The shape-and-effect system describes function parameters as l-value shapes, since actual parameters are in fact stored in stack variables. The returned value is an r-value expression. Function shapes carry a so-called *latent effect*, F, which accounts for the actions that (depending on the flow of control) *may* be performed during execution of the function. As an example, let us consider a simple function that adds one to an integer:

```
int plus1(int x) { return x+1; }
```

In this system, `plus1` may have shape $\mathsf{ref}_\rho\ \zeta \xrightarrow{read_\rho} \zeta$, where ζ is the shape of the input integer, and ρ is the region where the formal parameter `x` is stored. The use of the shape variable ζ indicates that the function is polymorphic on the shape of `x`; and at runtime this may be encoding a plain integer, or a pointer. The function simply reads the parameter, hence the effect $read_\rho$, and returns its value plus one. The result is an object of the same shape as `x`, either a plain integer or a pointer, hence the output shape is ζ as well.

L-value shapes

L-shapes denote *references* to either data or functions. Data (r-value) references have shape $\mathsf{ref}_\rho\ Z^R$, where ρ is a memory region, and Z^R is the shape of the objects that it holds. Data references are mutable unless declared `const`. Function references are immutable and have shape $\mathsf{ref}_\rho\ Z^F$. Whether a reference is mutable or immutable, that is enforced by the base C type system. If a reference ρ_1 holds a pointer to another reference ρ_2, as in $\mathsf{ref}_{\rho_1}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ Z$, we say that ρ_1 *points to* ρ_2. If two l-values a and b have the same shape $\mathsf{ref}_\rho\ Z$, we say that a and b are *aliases* (i.e., may denote the same object).

4.2 Shape-type compatibility.

A shape Z is *compatible* with a type T, written $Z \succ T$, if a runtime object of type T can be described by Z. There may be multiple shapes compatible with a given type. For instance, a value of type `int` may

$$\succ \subseteq \textsc{Shape} \times \textsc{Type}$$

$$\text{[Comp-Bot]}\quad \frac{T \in \{\texttt{void}, \texttt{float}, \texttt{double}\}}{\bot \succ T}$$

$$\text{[Comp-Int]}\quad \frac{T \in \{\texttt{char}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{long long}\}}{Z \succ T} \qquad \text{[Comp-Ptr]}\quad \frac{Z \succ T}{\mathsf{ptr}\ \mathsf{ref}_\rho\ Z \succ T*}$$

$$\text{[Comp-Ptr-Void]}\quad \frac{}{\mathsf{ptr}\ \mathsf{ref}_\rho\ Z \succ \texttt{void}*} \qquad \text{[Comp-Struct]}\quad \frac{\forall i \in [1,n].\ Z_i \succ T_i^A}{\texttt{struct}\ A_{|n}\ \{\ \overline{\mathsf{ref}_{\rho_i}\ Z_i\ x_i}\ \} \succ \texttt{struct}\ A}$$

$$\text{[Comp-Fun]}\quad \frac{\forall i \in [0,n].\ Z_i \succ T_i}{\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{F} Z_0 \succ T_1 \times \cdots \times T_n \rightarrow T_0}$$

Figure 4.2: Shape-type compatibility. (T_i^A denotes the type of the i-th member of the struct A.)

have shape $\bot$, if it is a plain integer number, or shape $\mathsf{ptr}\ \mathsf{ref}_\rho\ Z$ (for some Z) if it denotes a memory address. When the shape-and-effect system needs to *guess*[5] the shape of an identifier, it chooses a shape that is compatible with the declared type of the identifier. Figure 4.2 defines the compatibility relation between shapes and types.

Intuitively, shape-type compatibility requires that the given shape and type are structurally equivalent—as seen in the rules [Comp-Bot], [Comp-Ptr], and [Comp-Fun]. There are three small exceptions. First, any r-value shape is compatible with an integer type (rule [Comp-Int]). Similarly, any pointer shape is compatible with a pointer to `void` type (rule [Comp-Ptr-Void]). In other words, integers and pointers-to-`void` can point to arbitrary objects at runtime.

The third exception concerns struct shapes, which may refer only to an n-prefix of the members of the struct type (rule [Comp-Struct], see Sect. 4.3). For every member x_i in the n-prefix of A, the shape of x_i (Z_i)

[5]These guesses will become constraints in the inference algorithm, see Chapter 5.

$$\rhd \subseteq \textsc{Shape} \times \textsc{Shape}$$

$$\frac{}{Z \rhd Z}\ [\textsc{Cast-Refl}] \qquad \frac{}{Z \rhd \bot}\ [\textsc{Cast-Bot}] \qquad \frac{Z \rhd Z'}{\mathsf{ptr}\ \mathsf{ref}_\rho\ Z \rhd \mathsf{ptr}\ \mathsf{ref}_\rho\ Z'}\ [\textsc{Cast-Ptr}]$$

$$\frac{T_j^A \sim T_j^B \qquad Z_j \rhd Z'_j \qquad j \in [1,m] \qquad m \leq n}{\texttt{struct}\ A_{|n}\ \{\ \overline{\mathsf{ref}_{\rho_i}\ Z_i\ x_i}\ \} \rhd \texttt{struct}\ B_{|m}\ \{\ \overline{\mathsf{ref}_{\rho_i}\ Z'_i\ y_i}\ \}}\ [\textsc{Cast-Struct}]$$

$$\frac{Z'_i \rhd Z_i \ /\ i \in [1,n] \qquad F' \sqsupseteq F \qquad Z_0 \rhd Z'_0}{\overline{\mathsf{ref}_{\rho_i}\ Z_i} \xrightarrow{F} Z_0 \rhd \overline{\mathsf{ref}_{\rho_i}\ Z'_i} \xrightarrow{F'} Z'_0}\ [\textsc{Cast-Fun}]$$

Figure 4.3: Castable shapes. (T_i^A denotes the type of the i-th member of the struct A, and $\sim$ is the compatibility relation between C types.)

must be compatible with the declared type of x_i (T_i^A). Finally, a function shape is compatible with a function type (rule [Comp-Fun]), if the r-shapes and types of the formals, and the shape and type of the result, are all compatible. Neither the regions where the formals are stored, nor the latent effects, are considered for compatibility.

4.3 Shape casting

A shape Z can be *cast* to another shape Z', written $Z \rhd Z'$, if any object described by Z can be soundly manipulated as having shape Z'. The $\rhd$ relation is used to ensure that a type cast from type T (shape Z) to type T' (shape Z') does not circumvent the tracking of aliasing through memory regions. Roughly, it requires that any region of the target shape (Z') has a correspondence in the source shape (Z). Thus, after an assignment `x = (T) e`, any operation performed on `x` can be traced back to `e`.

Figure 4.3 defines the *castable* relation between shapes. For instance, a pointer `int**`, with shape $\mathsf{ptr}\ \mathsf{ref}_\rho\ \mathsf{ptr}\ \mathsf{ref}_{\rho'}\ Z$, can be cast to `float*` and manipulated as having shape $\mathsf{ptr}\ \mathsf{ref}_\rho\ \bot$ (rules [Cast-Ptr] and [Cast-Bot]). A cast in the opposite direction would not be sound, because a

memory access on ρ' could not be mapped back to ptr $\mathsf{ref}_\rho \perp$. The $\rhd$ relation for function shapes is co-variant on the latent effects and the result shape, and contra-variant on the shapes of the arguments (rule [Cast-Fun]) [LW94]. A function shape should be cast to another with the same number of arguments.

The tricky part is to handle casts involving struct types. In C, it is a common idiom to use casts between (pointers to) structs to work around the lack of *subtyping* (see example in Sect. 3.7, obs. 13). For instance, this allows that a 3D point (with fields `double x, y, z;`) can always be passed to a function expecting a 2D point (with fields `double x, y;`). Simply put, a struct shape A can be cast to another struct shape B, if both structs share a type-compatible m-prefix of members, and only members in that prefix will be accessed after the cast (rule [Cast-Struct]).

4.4 Environments and shape schemes

An environment Γ maps variables x to their reference shapes: $\Gamma(x) = \mathsf{ref}_\rho\ Z^R$; and function names f to *function shape schemes*:

$$\Gamma(f) = \forall\ \overline{v}.\ \mathsf{ref}_{\rho_0}\ (Z_1^L \times \cdots \times Z_n^L \xrightarrow{F} Z_0^R) \qquad \text{where } \rho_0 \notin \overline{v}$$

A function shape scheme is a function shape quantified over shape, region, and effect variables ($\overline{v}$) for which the function poses no constraints. We say that the function is *polymorphic* in such variables, which should occur free in the function shape (i.e., they are mentioned in $Z_1^L \times \cdots \times Z_n^L \xrightarrow{F} Z_0^R$). As such, these variables are parameters that can be appropriately instantiated at each call site. Let us consider again the function that adds one to an integer:

```
int plus1(int x) { return x+1; }
```

The shape of `plus1` can be generalized into $\forall\ \rho\zeta.\ \mathsf{ref}_\rho\ \zeta \xrightarrow{read_\rho} \zeta$, because `plus1` poses no constraint on the shape of the formal parameter `x`.

If F is of the form $F' \cup \varphi_0$ where $\varphi_0 \in \overline{v}$, we say that f is *effect-polymorphic*: the effect of f is extended by the instantiation of φ_0. (In our example, `plus1` can be made effect-polymorphic by introducing a new quantified variable φ_0, and setting its latent effects to $read_\rho \cup \varphi_0$.)

In general, it is unsound to generalize reference types [Tof90], but we can safely generalize function references because they are immutable. The memory region ρ_0 identifies the function; it is used to track calls to it through function pointers, and it cannot be generalized (thus $\rho_0 \notin \overline{v}$).

$$\vdash_{\mathrm{L}} \subseteq \textsc{Env} \times \textsc{Lval} \times \textsc{Shape} \times \textsc{Effects}$$

[Var]
$$\frac{\Gamma(x) = \mathsf{ref}_\rho\ Z}{\Gamma \vdash_{\mathrm{L}} x : \mathsf{ref}_\rho\ Z\ \&\ \emptyset}$$

[Deref]
$$\frac{\Gamma \vdash_{\mathrm{E}} E : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z\ \&\ F}{\Gamma \vdash_{\mathrm{L}} {\star}E : \mathsf{ref}_\rho\ Z\ \&\ F}$$

[Fun]
$$\frac{\Gamma(f) = \forall\ \overline{\zeta}\ \overline{\rho}\ \overline{\varphi}.\ \mathsf{ref}_{\rho_0}\ Z \qquad Z = Z_1^L \times \cdots \times Z_n^L\ \xrightarrow{F}\ Z_0^R}{\Gamma \vdash_{\mathrm{L}} f : \mathsf{ref}_{\rho_0}\ (Z[\overline{\zeta \mapsto Z'}][\overline{\rho \mapsto \rho'}][\overline{\varphi \mapsto F'}])\ \&\ \emptyset}$$

[Index]
$$\frac{\Gamma \vdash_{\mathrm{L}} L : \mathsf{ref}_{\rho_1}\ Z_1\ \&\ F_1 \qquad \Gamma \vdash_{\mathrm{E}} E : Z_2\ \&\ F_2}{\Gamma \vdash_{\mathrm{L}} L\,[E] : \mathsf{ref}_{\rho_1}\ Z_1\ \&\ F_1 \cup F_2}$$

[Member]
$$\frac{\Gamma \vdash_{\mathrm{L}} L : \mathsf{ref}_{\rho_0}\ \texttt{struct}\ A\ \{\ \overline{\mathsf{ref}_{\rho_i}\ Z_i\ x_i}\ \}\ \&\ F}{\Gamma \vdash_{\mathrm{L}} L.x_j : \mathsf{ref}_{\rho_j}\ Z_j\ \&\ F}$$

Figure 4.4: Typing of l-values.

4.5 Typing rules

This section presents the declarative typing rules for the proposed shape-and-effect system. This system assumes that the given terms already type-check under the C(IL) type system. The rules are then split according to CIL syntactic categories: l-values (Section 4.5.1), expressions (Section 4.5.2), instructions (Section 4.5.3), statements (Section 4.5.4), and global declarations (Section 4.5.5).

4.5.1 Typing of l-values

Judgment $\Gamma \vdash_{\mathrm{L}} L : \mathsf{ref}_\rho\ Z\ \&\ F$ (see Fig. 4.4) specifies that, under environment Γ, the l-value expression L has shape $\mathsf{ref}_\rho\ Z$, and evaluating it may result in effects F. An *l-value* always denotes a memory location, therefore has shape $\mathsf{ref}_\rho\ Z$. Note that, operationally, an l-value is just a pointer expression. Evaluating an l-value computes the memory address of the target memory cell, but this does not necessarily cause any memory access. I discuss the rules of Fig. 4.4 in order below.

The shape of a variable x is obtained directly from the environment (rule [Var]). Pointer dereferencing proceeds by evaluating an expression E, which produces F side effects, and obtaining the reference object associated to the resulting memory address (rule [Deref]). Dereferencing has no effects by itself, when used in an l-value context, that is why the following holds: `&*NULL == NULL` (i.e., the `NULL` memory address is not really being accessed). The memory address is effectively accessed when the l-value is used in an r-value context (see rule [Lval], Fig. 4.5).

The shape of a function name f is obtained by appropriately instantiating its shape scheme (rule [Fun]). This instance is generated by substituting quantified variables with concrete shapes, regions and effects. In a typing derivation, these will depend on the calling context: the actual parameters passed to the function, and the expected shape of the function's return value in that context.

Subscript expressions (also, array indexing expressions) take a *base* l-value expression L, and an *index* expression E, and compute a reference to the E-element of L (rule [Index]). Unlike in C, in CIL, L must be of type array, and these expressions are intended for array indexing only. Operationally, L is the base memory address, E is an offset, and the result is obtained through pointer arithmetic. Here, array shapes are flattened (cf. Sect. 4.1.3).

Member access expressions return a reference to a member in a structured object (rule [Member]). For pragmatic reasons, this system unsoundly treats regions ρ_0 and ρ_i as if they were independent (cf. Sect. 4.1.3). In reality, memory locations ρ_i are given as an offset to the location of the struct object ρ_0. (That is why this rule does not introduce the effect $read_{\rho_0}$, there is no such a read operation.) Note that in CIL, an expression E`->`x is always represented as `(*`E`).`x.

4.5.2 Typing of expressions

Expressions denote *values*, and have either r-value or function shapes. CIL expressions are *side-effect free*, but evaluating an expression may still involve reading memory locations. Such reads are recorded as effects in this system. Figure 4.5 introduces the typing rules for expressions. The judgment $\Gamma \vdash_{\mathrm{E}} E : Z \;\&\; F$ specifies that, in the environment Γ, evaluating the expression E results in a value of shape Z, and may produce F effects.

$$\vdash_{\mathrm{E}} \subseteq \textsc{Env} \times \textsc{Exp} \times \textsc{Shape} \times \textsc{Effect}$$

[Const-Bot]
$$\frac{\mathrm{typeof}(c) \in \{\texttt{float}, \texttt{double}\}}{\Gamma \vdash_{\mathrm{E}} c : \bot \;\&\; \emptyset}$$

[Const-Str]
$$\frac{\mathrm{typeof}(str) = \texttt{char}\star}{\Gamma \vdash_{\mathrm{E}} str : \mathsf{ptr}\ \mathsf{ref}_\rho\ \bot \;\&\; \emptyset}$$

[Const-Int]
$$\frac{\mathrm{typeof}(i) \in \{\texttt{char}, \texttt{int}, \texttt{short}, \texttt{long}, \texttt{long long}\}}{\Gamma \vdash_{\mathrm{E}} i : Z \;\&\; \emptyset}$$

[Lval]
$$\frac{\Gamma \vdash_{\mathrm{L}} L : \mathsf{ref}_\rho\ Z \;\&\; F}{\Gamma \vdash_{\mathrm{E}} L : Z \;\&\; F \cup \{read_\rho\}}$$

[Addr]
$$\frac{\Gamma \vdash_{\mathrm{L}} L : \mathsf{ref}_\rho\ Z \;\&\; F}{\Gamma \vdash_{\mathrm{E}} \&L : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z \;\&\; F}$$

[Neg]
$$\frac{\Gamma \vdash_{\mathrm{E}} E : Z \;\&\; F \qquad \ominus \in \{-, \sim\}}{\Gamma \vdash_{\mathrm{E}} \ominus E : Z \;\&\; F}$$

[Not]
$$\frac{\Gamma \vdash_{\mathrm{E}} E : Z \;\&\; F}{\Gamma \vdash_{\mathrm{E}} !E : \bot \;\&\; F}$$

[Int-A]
$$\frac{\otimes \in \{+, -, \star, /, \%, \&, \hat{}, |, \ll, \gg\} \qquad \Gamma \vdash_{\mathrm{E}} E_1 : Z \;\&\; F_1 \qquad \Gamma \vdash_{\mathrm{E}} E_2 : Z \;\&\; F_2}{\Gamma \vdash_{\mathrm{E}} E_1 \otimes E_2 : Z \;\&\; F_1 \cup F_2}$$

[Bool-A]
$$\frac{\Gamma \vdash_{\mathrm{E}} E_1 : Z_1 \;\&\; F_1 \qquad \Gamma \vdash_{\mathrm{E}} E_2 : Z_2 \;\&\; F_2 \qquad \otimes \in \{\&\&, ||\}}{\Gamma \vdash_{\mathrm{E}} E_1 \otimes E_2 : \bot \;\&\; F_1 \cup F_2}$$

[Cmp]
$$\frac{\preceq \in \{<, >, <=, >=, ==, !=\} \qquad \Gamma \vdash_{\mathrm{E}} E_1 : Z_1 \;\&\; F_1 \qquad \Gamma \vdash_{\mathrm{E}} E_2 : Z_2 \;\&\; F_2}{\Gamma \vdash_{\mathrm{E}} E_1 \preceq E_2 : \bot \;\&\; F_1 \cup F_2}$$

[Question]
$$\frac{\Gamma \vdash_{\mathrm{E}} E_1 : Z_1 \;\&\; F_1 \qquad \Gamma \vdash_{\mathrm{E}} E_2 : Z \;\&\; F_2 \qquad \Gamma \vdash_{\mathrm{E}} E_3 : Z \;\&\; F_3}{\Gamma \vdash_{\mathrm{E}} (E_1)\ ?\ E_2\ :\ E_3 : Z \;\&\; F_1 \cup F_2 \cup F_3}$$

[Cast]
$$\frac{\Gamma \vdash_{\mathrm{E}} E : Z \;\&\; F \qquad Z' \succ T \qquad Z \rhd Z'}{\Gamma \vdash_{\mathrm{E}} (T)\, E : Z' \;\&\; F}$$

Figure 4.5: Typing of expressions. (We use *typeof* to refer to the base C(IL) type system.)

Constants. Constants are divided into three groups. Constants of C types that cannot be used to encode pointers have $\perp$ shape (rule [Const-Bot]). String literals have pointer shapes (rule [Const-Str]). These literals are assigned to some context-dependent region ρ. Constants of integer types (`char` or larger) may encode pointers, and thus can have arbitrary r-value shapes (rule [Const-Int]). The shape of an integer is unknown *a priori*, and depends on how this integer value is used by the program. This is why the system allows an arbitrary shape Z for integer constants, which will be constrained by the context during type inference. For instance, in a expression like `ptr + 1`, where `ptr` is a pointer variable, the constant `1` would be given the same pointer shape as `ptr`.

L-values. Fetching the content of (the memory location denoted by) an l-value, results in an expression of the expected shape, and produces a *read* effect on the corresponding memory region (rule [Lval]). (In C there is no explicit operator to read from a reference.) Given an l-value (i.e., a reference), the *address-of* operator (`&`) obtains the address of the corresponding memory cell (rule [Addr]). This operation is side-effect free because no memory location is being accessed—a pointer is the r-value representation of an l-value.

Expressions `sizeof()` and `alignof()`. The `sizeof`() operator computes the storage size of an expression or type in bytes, whereas the `alignof`() operator computes the memory alignment requirements.[6] (Rules for these two operators were omitted in Fig. 4.5 for layout reasons, but are given below.) Expressions `sizeof`(T) and `alignof`(T) are statically resolved to constants, and hence produce no effects at runtime (rules [Sizeof-T] and [Alignof-T]). With the exception of `sizeof`(T`[`E`]`), a variable length array type, which is interpreted as the value of E times the size of T (i.e., `(`E`)*sizeof(`T`)`), and requires the evaluation of the expression E at runtime ([Sizeof-A]). Expressions `sizeof`(E) and `alignof`(E) produce no effects either, because only the type of the expression is considered (rules [Sizeof-E] and [Alignof-E]). The result of any these operations can be used to compute a memory address, and for

[6]The alignment represents the number of bytes between successive addresses at which objects of a given type can be allocated. The alignment depends on the underlying computer architecture, and it is chosen to maximize the efficiency of memory accesses.

that reason it can take any arbitrary shape, depending on the context. This system does not check any of the restrictions that ANSI C imposes on the arguments of `sizeof`() and `alignof`()—the base type system does.

$$\text{[Sizeof-T]}\ \frac{T \text{ is not an array type}}{\Gamma \vdash_{\mathrm{E}} \texttt{sizeof}(T) : Z \,\&\, \emptyset} \qquad \text{[Sizeof-A]}\ \frac{\Gamma \vdash_{\mathrm{E}} E : Z \,\&\, F}{\Gamma \vdash_{\mathrm{E}} \texttt{sizeof}(T\texttt{[}E\texttt{]}) : Z \,\&\, F}$$

$$\text{[Sizeof-E]}\ \frac{}{\Gamma \vdash_{\mathrm{E}} \texttt{sizeof}(E) : Z \,\&\, \emptyset} \qquad \text{[Alignof-T]}\ \frac{}{\Gamma \vdash_{\mathrm{E}} \texttt{alignof}(T) : Z \,\&\, \emptyset}$$

$$\text{[Alignof-E]}\ \frac{}{\Gamma \vdash_{\mathrm{E}} \texttt{alignof}(E) : Z \,\&\, \emptyset}$$

Integer and bitwise operations. The effect of any integer arithmetic or bitwise operation, is the combined effect of evaluating its operands (rules [Neg] and [Int-A]). These operations can be used to perform pointer arithmetic, either directly, or indirectly by masquerading pointers as integers. For binary operations, both operands must have the same shape. Thus, arithmetic between pointers to different memory regions, or with incompatible shapes, is disallowed. For instance, in a expression like `ptr + 1`, where `ptr` is a pointer variable, the constant `1` would be given the same shape as `ptr`. The result pointer has the same shape, and belongs to the same region, as the operands. Note that pointer arithmetic is only well-defined in a few specific cases. This system does not prohibit any arithmetic operations on pointers that would lead to undefined behaviors, but in that case the aliasing information may be unsound. Unfortunately, precise reasoning about pointer arithmetic requires automated theorem proving, which would seriously impact performance.

Pointer operations. CIL distinguishes the well-defined cases of adding an offset to a memory address, and subtracting two memory addresses (to obtain an offset). For this, CIL relies on the C types of expression, and therefore it does not recognize cases of pointer arithmetic masquerade as integer arithmetic. Even though rules [Neg] and [Int-A] already handle arithmetic on pointers, we can also consider the explicit pointer operations separately. This does not require to assign a pointer shape

to *offset* integer constants, and may improve precision if the same offset is used in computing addresses to different regions. The typing rules would be:

$$\text{[Plus-A]}\quad \frac{\Gamma \vdash_{\mathrm{E}} E_1 : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z_1\ \&\ F_1 \qquad \Gamma \vdash_{\mathrm{E}} E_2 : Z_2\ \&\ F_2}{\Gamma \vdash_{\mathrm{E}} E_1{+}E_2 : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z_1\ \&\ F_1 \cup F_2}$$

$$\text{[Minus-PP]}\quad \frac{\Gamma \vdash_{\mathrm{E}} E_1 : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z\ \&\ F_1 \qquad \Gamma \vdash_{\mathrm{E}} E_2 : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z\ \&\ F_2}{\Gamma \vdash_{\mathrm{E}} E_1{-}E_2 : \bot\ \&\ F_1 \cup F_2}$$

Logical operations and comparisons. The effect of any logical operation or comparison, is the combined effect of evaluating its operands (rules [Not] and [Bool-A]). These rules pose no requirement on the shapes of the operands. The result is Boolean and always has shape $\bot$.

Conditionals. Both alternatives of a conditional expression contribute to the effect of the entire expression (rule [Question]). Note that, in a particular execution, either E_1 or E_2 will be evaluated, but not both. The union of all three effects is a flow-insensitive *over*-approximation. Both branches shall have the same shape, which is also the shape of the overall expression.

Type casts. An expression can be cast to a type T if, for some shape Z' that is compatible with T, the shape of the expression Z can be cast to Z' (see rule [Cast] and Fig. 4.3). The purpose of this rule is to discern which type casts are handled—mostly—soundly (cf. Sect. 4.3), even though in practice any cast is allowed, see Sect. 5.5. Most type casts used to work around type genericity or struct subtyping are correctly handled by this system.

4.5.3 Typing of instructions

CIL instructions denote basic program steps without control flow. They correspond to C side-effectful expressions: assignments and function calls. Figure 4.6 shows the typing rules for instructions. The judgment $\Gamma \vdash_{\mathrm{I}} I : Z\ \&\ F$ specifies that, in the environment Γ, evaluating the instruction I results in a value of shape Z, and produces F effects. (CIL

$$\vdash_{\mathrm{I}} \subseteq \textsc{Env} \times \textsc{Instr} \times \textsc{Shape} \times \textsc{Effect}$$

[Set-Exp]
$$\frac{\Gamma \vdash_{\mathrm{L}} L : \mathsf{ref}_{\rho}\ Z\ \&\ F_1 \qquad \Gamma \vdash_{\mathrm{E}} E : Z\ \&\ F_2}{\Gamma \vdash_{\mathrm{I}} L\ =\ E : Z\ \&\ F_1 \cup F_2 \cup \{write_{\rho}\}}$$

[Call]
$$\frac{\Gamma \vdash_{\mathrm{E}} L_f : \mathsf{ref}_{\rho_0}\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{F'} Z_0)\ \&\ F_0 \qquad \Gamma \vdash_{\mathrm{E}} E_i : Z_i\ \&\ F_i\ /\ i \in [1,n]}{\Gamma \vdash_{\mathrm{I}} L_f(E_1, \cdots, E_n) : Z_0\ \&\ F_0 \cup (\bigcup_{i \in [1,n]} F_i) \cup \{call_{\rho_0}\} \cup F'}$$

[Set-Call]
$$\frac{\Gamma \vdash_{\mathrm{L}} L_x : \mathsf{ref}_{\rho}\ Z\ \&\ F \qquad \Gamma \vdash_{\mathrm{I}} L_f(E_1, \cdots, E_n) : Z\ \&\ F'}{\Gamma \vdash_{\mathrm{I}} L_x = L_f(E_1, \cdots, E_n) : Z\ \&\ F \cup F' \cup \{write_{\rho}\}}$$

Figure 4.6: Typing of instructions.

instructions do not produce values when evaluated, but assuming that they do allows for a more compact formulation of rule [Set-Call], see below.)

Assignment instructions come in two forms depending on whether the right-hand side is an expression (rule [Set-Exp]), or a function call (rule [Set-Call]). In any case, an assignment instruction writes the result of evaluating the right-hand side into the memory location denoted by the l-value on the left. In addition to the effects of evaluating both sides, the system records the effect of writing to memory region ρ. Left- and right-hand side must have the same shape. Hence, a pointer assignment like `ptr = &x;` implies that `*ptr` and `x` alias (as shapes include region annotations).

Function application takes a function reference and a tuple of arguments of the right shape (rule [Call]). Calls to functions are recorded with calling $call_{\rho_0}$ effects, where region ρ_0 identifies the callee. The latent effects F' of the function are recorded as potential side-effects of the invocation. At run-time, a particular application may perform only a subset of these effects, but shall not perform any effect outside F'.

4.5.4 Typing of statements

Statements add control flow to CIL instructions. Figure 4.7 shows the typing rules for statements. The judgment $\Gamma \vdash_S^{\Downarrow Z} S \ \& \ F$ specifies that, in the environment Γ, and in the body of a function that returns values of shape Z, the statement S is valid, and its evaluation *may* produce effects F. Note that a flow-insensitive analysis like this one, basically ignores control flow. The effects of a statement are computed as the sum of the effects resulting from the evaluation of all its sub-expressions and sub-statements.

Basic statements. A semicolon converts an instruction into an statement (rule [Instr]). The value of the instruction is ignored (it may have been stored in memory), but the effects are propagated. Every return value must match the result shape of the enclosing function (see rule [Return-E], and rule [Fun-Def] in Fig. 4.8). No restriction applies to functions returning `void` (rule [Return]). Statements can be labeled (rule [Label]). Unstructured control-flow constructs are basically irrelevant for this system and have no effects (rules [Goto], [Break], and [Continue]).

Composite statements. The effect of a branching statement is the sum of the effects of all its branches (rules [If] and [Switch]). This is an over-approximation of the effects at runtime. Similarly, the effects of a loop statement[7] are those of the loop body, regardless of how many times the body is executed, if executed at all (rule [Loop]). Finally, statements can be grouped to be executed sequentially (rule [Block]).

4.5.5 Typing of globals

A C translation unit is made of several global declarations and definitions (for short, *globals*). Figure 4.8 shows the typing rules for globals. The judgment $\Gamma \vdash_G G \ \& \ \Gamma'$ specifies that, in the environment Γ, the global G is valid, and transforms the environment into Γ'. (Note that C globals have no runtime effects, in particular, the initialization of global variables occurs at compile-time.) If the translation unit depends on externally defined variables or functions, these shall already be present in the environment at the time of typing.

[7] In CIL, every C loop is encoded with an infinite loop and the use of unstructured control-flow (`goto`, etc).

$$\vdash_{\mathrm{S}} \subseteq \textsc{Env} \times \textsc{Shape} \times \textsc{Stmt} \times \textsc{Effect}$$

[Instr]
$$\frac{\Gamma \vdash_{\mathrm{I}} I : Z' \,\&\, F}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} I; \,\&\, F}$$

[Return]
$$\frac{}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow \bot} \texttt{return;} \,\&\, \emptyset}$$

[Return-E]
$$\frac{\Gamma \vdash_{\mathrm{E}} E : Z \,\&\, F}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} \texttt{return } E\texttt{;} \,\&\, F}$$

[Label]
$$\frac{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} S \,\&\, F}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} l\texttt{:} \quad S\texttt{;} \,\&\, F}$$

[Goto]
$$\frac{}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} \texttt{goto } l\texttt{;} \,\&\, \emptyset}$$

[Break]
$$\frac{}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} \texttt{break;} \,\&\, \emptyset}$$

[Continue]
$$\frac{}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} \texttt{continue;} \,\&\, \emptyset}$$

[If]
$$\frac{\Gamma \vdash_{\mathrm{E}} E : Z_0 \,\&\, F_0 \qquad \Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} S_1 \,\&\, F_1 \qquad \Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} S_2 \,\&\, F_2}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} \texttt{if (}E\texttt{)} \;\; S_1 \;\; S_2 \,\&\, F_0 \cup F_1 \cup F_2}$$

[Switch]
$$\frac{\Gamma \vdash_{\mathrm{E}} E : Z_0 \,\&\, F_0 \qquad \forall i \in [1,n].\ \Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} S_i \,\&\, F_i}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} \texttt{switch (}E\texttt{) \{ } S_1 \cdots S_n \texttt{ \}} \,\&\, F_0 \cup \left(\bigcup_{i \in [1,n]} F_i \right)}$$

[Loop]
$$\frac{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} S \,\&\, F}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} \texttt{while (1) } S \,\&\, F}$$

[Block]
$$\frac{\forall i \in [1,n].\ \Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} S_i \,\&\, F_i}{\Gamma \vdash_{\mathrm{S}}^{\Downarrow Z} \{S_1 \cdots S_n\} \,\&\, \bigcup_{i \in [1,n]} F_i}$$

Figure 4.7: Typing of statements.

At this point, let us focus on the declaration and definition of variables and functions. For presentation purposes, let us also assume that there is a single declaration or definition for each variable, and a single definition for each function; and that there are no mutually recursive definitions. Multiple variable or function prototype declarations can be grouped and handled as one. Typing mutually recursive function definitions requires a simple yet convoluted extension of rule [Fun-Def], so

$$\vdash_{\mathrm{G}} \subseteq \textsc{Env} \times \textsc{Global} \times \textsc{Env}$$

[Var-Decl]
$$\frac{\Gamma' = \Gamma; x : Z \qquad Z \succ T}{\Gamma \vdash_{\mathrm{G}} T\ x;\ \&\ \Gamma'}$$

[Var-Def]
$$\frac{\Gamma \vdash_{\mathrm{E}} E : Z\ \&\ F \qquad \Gamma' = \Gamma; x : Z \qquad Z \succ T}{\Gamma \vdash_{\mathrm{G}} T\ \ x\ =\ E;\ \&\ \Gamma'}$$

[Fun-Def]
$$\frac{\begin{array}{c} Z_f = \mathsf{ref}_{\rho_0}\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n\ \xrightarrow{F}\ Z_0) \\ \Gamma; f : \forall\ .\ Z_f; x_1 : \mathsf{ref}_{\rho_1}\ Z_1; \cdots; x_m : \mathsf{ref}_{\rho_m}\ Z_m \vdash_{\mathrm{S}}^{\Downarrow Z_0} S\ \&\ F_0 \\ Z_i \succ T_i\ /\ i \in [0,m] \qquad F \sqsupseteq F_0 \\ \Gamma' = \forall\ \overline{v}.\ \mathsf{ref}_{\rho_0}\ Z_f \qquad \overline{v} = \mathrm{FTV}(Z_f) \setminus (\mathrm{FTV}(\Gamma) \cup \{\rho_0\}) \end{array}}{\Gamma \vdash_{\mathrm{G}} T_0\ \ f\,(T_1\ \ x_1, \cdots, T_n\ \ x_n)\ \ \{\ \ T_{n+1}\ x_{n+1}; \cdots T_m\ x_m;\ S\ \ \}\ \&\ \Gamma'}$$

[Tr-Unit]
$$\frac{\Gamma_{i-1} \vdash_{\mathrm{G}} G_i\ \&\ \Gamma_i \qquad i \in [1,n]}{\Gamma_0 \vdash_{\mathrm{G}} G_1 \cdots G_n\ \&\ \Gamma_n}$$

Figure 4.8: Typing of globals.

that the functions in a recursive group are typed all together.

Variables are given a shape that is compatible with their declared type, and introduced into the environment (rules [Var-Decl] and [Var-Def]). In a variable definition, the effects of evaluating the initializer are ignored. Note that, in C, global initializers must be constant expressions, and the initialization of global variables introduces no runtime effects. (Again, for presentation purposes, I have obviated array and struct initializers.)

For a function definition f, its body S is typed under an extended environment, where each parameter or local variable x_i is given a shape compatible with its declared type (rule [Fun-Def]). This environment includes f itself, with a monomorphic shape, allowing for (monomorphic) recursion. The latent effects of f must be a superset of those resulting of evaluating its body. Finally, in the new environment, the shape of f

is generalized over those variables $\overline{v}$, that are unique to f's shape and hence do not occur free in Γ.

A translation unit is typed by typing each one of its globals sequentially, in order of appearance (rule [Tr-Unit]). Each global produces a new environment that is used to type the subsequent ones.

4.6 Soundiness

In order to be simple and yet useful, the shape and effect system had to be unsound. Unsoundness affects mainly (but not only) the treatment of structs and pointer arithmetic. This is a necessary trade-off for a bug-finding technique [LSS+15]. The biggest challenge, as it is often the case in static analysis, is to keep track of aliasing in an unsafe language like C. The language of shapes is of great help to deal with this problem, but that alone is not enough to deal with the use of certain workarounds involving type casts, pointer arithmetic, and collections of data elements (e.g., arrays and linked lists). Table 4.1 summaries these trade-offs.

In particular, this system has been designed to handle struct objects and casts between struct types precisely, but not soundly. The need for a precise treatment of structures has been identified by previous work [Ste96a, YHR99], and in the qualitative study of Linux bugs which is part of this thesis (cf. Sect. 3.7). I could also confirm this experimentally, when running early versions of EBA on Linux code. One simply cannot

Table 4.1: Design compromises of the shape-and-effect system.

Decision	**Pros**	**Cons**
Flow-insensitive analysis	efficient	imprecise
Polymorphic shapes, regions and effects	precise	complex
Struct members are treated as separate variables	precise	unsound
Collections of data elements (e.g., arrays, linked lists) have flat shapes	sound	imprecise
Dynamic memory allocations at a given location are conflated.	sound	imprecise

expect to get anything useful from an analysis that is not precise on structures. With a decent handling of structures, EBA finds a dozen bugs in Linux, with a low number of false alarms, despite handling pointer arithmetic and data collections naively (see Chapter 7).

There are still some tricky uses of structs that this system does not handle satisfactorily. This is, for instance, the case of recovering a pointer to a struct object from a pointer to one of its fields. In C, an expression of type `int*` may not only be a pointer to an `int`, but may also be a pointer to an `int` field of some struct object. If one knows to which field, and of which struct, a pointer points to, it is possible to compute the memory address of the container struct object by means of the `offsetof(type,member)` macro.[8] The Linux macro `container_of(ptr,type,member)` does exactly this: given a pointer `ptr` to the member `member` of an object of struct-type `type`, it computes the base pointer of the container struct.

The above trick is used to implement a form of object-downcast in C. For instance, in Linux, device drivers are maintained in a tree-like data structure, where data elements are pointers to `struct device` objects. Each of these pointers is typically embedded in a driver-specific object (i.e., another structured object with a `struct device` member) that maintains the state of the device driver. While the driver API requires that certain functions simply receive a pointer to an `struct device` object as input, the drivers' code can use `container_of` to obtain a pointer to the container driver-specific object from it. This shape-and-effect system does not offer a good solution to handle this idiom. One approximation could be to introduce a new kind of *projection shapes*, that could describe pointers to struct members, while retaining information about the container objects.

[8]The expression `offsetof(struct` A`,`x`)` returns the offset in bytes of the member x with respect to the base address of an `struct` A object.

Chapter 5

Shape-Region and Effect Inference for C(IL)

The shape-and-effect system has been presented in a declarative fashion, which is useful for understanding and reasoning about it. Such formulation, however, is not directly executable since the typing rules do not constitute an algorithm. The typing rules often make *guesses* (i.e., non-deterministic choices) of regions, shapes, and effects. For instance, the l-value rule [FUN] (cf. Fig.4.4) picks arbitrary regions, shapes, and sets of effects in order to instantiate a function shape scheme.

This chapter describes how to derive an inference algorithm for the shape-and-effect system of Chapter 4, following the recipe given by Talpin and Jouvelot in [JT93]. Every set of rules presented here constitutes an algorithm (rules must be applied left to right and top to bottom). This algorithm shall [1] infer *principal types* and *minimum sets of effects*. The standard way of doing this is to replace all guesses with *fresh* variables, and introduce a set of constraints on the values that each one of these variables can take. This is known as *constraint-based type inference*.

5.1 Unification

Whenever a typing rule requires two (a priori different) shapes to be the same, this translates into an equality constraint in the inference algorithm. For instance, rule [INT-A] (cf. Fig. 4.5), for typing integer arithmetic expressions, requires that both of its operands have the same

[1] I have not proven it, see Sect. 5.6

$$\rightsquigarrow \;:\; \textsc{Shape} \times \textsc{Shape} \to \textsc{Subst}$$

[Unif-Bot]
$$\frac{}{\bot \rightsquigarrow \bot = \emptyset}$$

[Unif-Ptr-Bot]
$$\frac{}{\mathsf{ptr}\ \mathsf{ref}_\rho\ Z \rightsquigarrow \bot = \emptyset}$$

[Unif-Var-Bot]
$$\frac{}{\zeta \rightsquigarrow \bot = \emptyset}$$

[Unif-Var-L]
$$\frac{\zeta \notin \mathrm{FTV}(Z)}{\zeta \rightsquigarrow Z = \{\zeta \mapsto Z\}}$$

[Unif-Var-R]
$$\frac{\zeta \notin \mathrm{FTV}(Z)}{Z \rightsquigarrow \zeta = \{\zeta \mapsto Z\}}$$

[Unif-Ptr]
$$\frac{Z_1 \rightsquigarrow Z_2 = \theta}{\mathsf{ptr}\ \mathsf{ref}_{\rho_1}\ Z_1 \rightsquigarrow \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ Z_2 = \{\rho_2 \mapsto \rho_1\}\theta}$$

[Unif-Struct]
$$\frac{T_k^A \sim T_k^B \qquad \theta_k^\rho = \{\rho_k' \mapsto \rho_k\} \qquad k \in [1,n] \qquad \theta_k^\rho\theta_{k-1}\cdots\theta_1 Z_k \rightsquigarrow \theta_k^\rho\theta_{k-1}\cdots\theta_1 Z_k' = \theta_k}{\texttt{struct}\ A\ \{\ \overline{\mathsf{ref}_{\rho_i}\ Z_i\ x_i}\ \} \rightsquigarrow \texttt{struct}\ B_{|n}\ \{\ \overline{\mathsf{ref}_{\rho_j'}\ Z_j'\ y_j}\ \} = \theta_n\cdots\theta_1}$$

[Unif-Fun]
$$\frac{\theta_k^\rho = \{\rho_k' \mapsto \rho_k\} \qquad \theta_k^\rho\theta_{k-1}\cdots\theta_1 Z_k \rightsquigarrow \theta_k^\rho\theta_{k-1}\cdots\theta_1 Z_k' = \theta_k \qquad k \in [1,n] \qquad Z_0 \rightsquigarrow Z_0' = \theta_0 \qquad \theta' = \theta_n\cdots\theta_0 \qquad \theta = \{\theta'\varphi' \mapsto \theta'\varphi\}\theta'}{\mathsf{ref}_{\rho_1}\ Z_1 \times\cdots\times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\varphi} Z_0 \rightsquigarrow \mathsf{ref}_{\rho_1'}\ Z_1' \times\cdots\times \mathsf{ref}_{\rho_n'}\ Z_n' \xrightarrow{\varphi'} Z_0' = \theta}$$

Figure 5.1: Shape unification.

shape. The inference algorithm solves these equations by using unification [Rob65], which takes two shapes and either fails,[2] or produces a substitution mapping type variables to regions, shapes and sets of effects—depending on the kind of variable.

Another source of constraints are type casts, which require *mapping* the memory regions of one shape to another (potentially incompatible) shape. This mapping is defined by the $\rhd$ relation (cf. Sect. 4.3). Note that the only typing rule that introduces $\rhd$-constraints is [Cast] (cf. Fig. 4.5), because in CIL all type conversions are explicit. The inference algorithm solves these inequalities in a similar fashion than the equalities, with an

[2]Unification failures are handled specially, see Sect. 5.5.

algorithm derived from the $\rhd$ relation. In fact, this algorithm, which we will call $\rhd$-*unification*, embeds shape unification and is used to handle equality constraints as well.

Figure 5.1 presents the $\rhd$-unification algorithm. In the inference rules, both $Z_1 = Z_2$ and $Z_1 \rhd Z_2$ constraints are converted into $Z_1 \rightsquigarrow Z_2 = \theta$. The $\rightsquigarrow$ operator $\rhd$-unifies shapes Z_1 and Z_2, producing a substitution θ, so that $\theta Z_1 \rhd \theta Z_2$. Any non-struct shape trivially $\rhd$-unifies with $\bot$ (rules [Unif-Bot], [Unif-Ptr-Bot], and [Unif-Var-Bot].) The $\rhd$-unification of a shape variable ζ with a shape Z (and vice-versa) simply maps ζ to Z (rules [Unif-Var-L] and [Unif-Var-R]). Yet, to avoid cycles, it is required that ζ is not free in Z. For example, $\bot \rightsquigarrow \zeta = \{\zeta \mapsto \bot\}$ (rule [Unif-Var-R]), but $\zeta \rightsquigarrow \mathsf{ptr}\ \mathsf{ref}_\rho\ \zeta =$ *failure* because $\zeta \in \mathrm{FTV}(\mathsf{ptr}\ \mathsf{ref}_\rho\ \zeta)$.

The $\rhd$-unification of two pointer shapes *merges* the regions in memory where they point to (effectively recording them as aliases), and recursively $\rhd$-unifies the "pointees" (rule [Unif-Ptr]). For example, $\mathsf{ptr}\ \mathsf{ref}_{\rho_1} \bot \rightsquigarrow \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ \zeta = \{\zeta \mapsto \bot, \rho_2 \mapsto \rho_1\}$ (rules [Unif-Ptr] and [Unif-Var-R]). Rules for struct ([Unif-Struct]) and function ([Unif-Fun]) shapes are, despite convoluted, systematically derived from the corresponding $\rhd$-rules [Cast-Struct] and [Cast-Fun] of Fig. 4.3. The astute reader may have noticed that in rule [Cast-Fun] the latent effects of both function shapes are just effect variables, rather than arbitrary sets of effects, this will be explained in Sect. 5.3.

5.2 Most general shape

In the declarative type system, whenever a shape Z had to be chosen to describe an arbitrary object of type T, we required Z to be compatible with T ($Z \succ T$, cf. Sect. 4.2). The inference algorithm introduces the notion of *most general shape*, which is derived from the definitions of the $\succ$ and $\rhd$ relations (cf. figs. 4.2 and 4.3). The most general shape of a type T, written shape-of(T), is a shape Z that is compatible with T (i.e., $Z \succ T$), and for any other Z' that is also compatible with T, there exist a substitution θ such that $Z \rightsquigarrow Z' = \theta$.

Figure 5.2 shows the algorithmic rules for shape-of$()$. These rules shall[3] obtain the most-general shape of a given type. Essentially, whenever a type is $\succ$-compatible with two or more shapes, these rules prefer

[3]But I have not proven it.

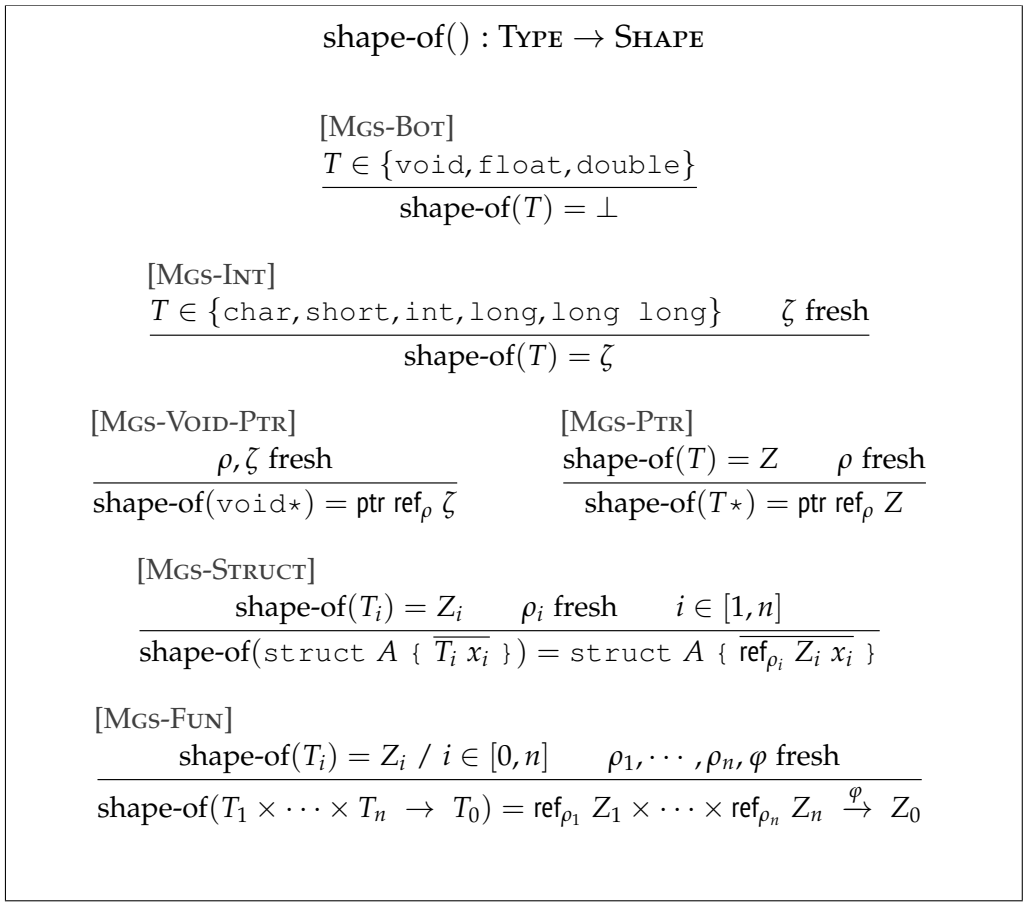


Figure 5.2: Most general shape.

the most general one with respect to $\rhd$. Hence, fresh shape variables are preferred whenever possible, since they can be instantiated with arbitrary r-shapes (cf. Fig. 5.1); and pointer shapes are preferred over $\bot$ (rule [Cast-Bot] of Fig. 4.3). For instance, according to rule [Comp-Int] of $\succ$, the shape of `int` could be $\bot$, ptr $\mathsf{ref}_\rho\ Z$, and ζ—for arbitrary Z and ζ. Rule [Mgs-Int] specifies that the default shape assigned to an object of type `int` shall be a fresh shape variable ζ, the most general of the three.

5.3 Subeffecting constraints

Besides equality and cast constraints between shapes, which are solved by $\rhd$-unification, there are also equality and subset constraints between

sets of effects, which are introduced by rule [Fun-Def] (cf. Fig. 4.8). Solving these two types of constraints on sets of effects will be reduced to solving a system of subeffecting constraints [TJ92].

A *subeffecting constraint* written $\varphi \sqsupseteq F$ specifies that any solution for the variable φ must include at least the effects F. A *system of subeffecting constraints*, denoted by κ, is a set of subeffecting constraints where the left hand sides of the inequations are distinct. (Note that $\{\varphi \sqsupseteq F_1; \varphi \sqsupseteq F_2\}$ can be reduced to $\{\varphi \sqsupseteq F_1 \sqcup F_2\}$.) The *restriction* of a constraint system κ on the effect variables $\overline{v}$ is defined as $\kappa_{\overline{v}} = \{\varphi \sqsupseteq F \in \kappa \mid \varphi \in \overline{v}\}$.

A constraint system κ, by construction, always admits at least one solution [TJ92]. A solution of κ is a substitution θ such that $\theta\varphi \sqsupseteq \theta F$ holds for every $\varphi \sqsupseteq F \in \kappa$ The principal model of a system κ, written $\overline{\kappa}$, is inductively defined as $\overline{\{\}} = \emptyset$, and $\overline{\kappa' \cup \{\varphi \sqsupseteq F\}} = \{\varphi \mapsto F'\}\overline{\kappa'}$ where $F' = \overline{\kappa'}(\varphi \sqcup F)$. For any solution θ of a system κ, there exist θ' such that $\theta = \theta'\overline{\kappa}$. The minimal solution of κ, denoted by $\mathrm{Min}(\kappa)$, is computed as $\mathrm{Min}(\{\}) = \emptyset$, and $\mathrm{Min}(\kappa' \cup \{\varphi \sqsupseteq F\}) = \{\varphi \mapsto \theta' F \setminus \{\varphi\}\}\theta'$ where $\theta' = \mathrm{Min}(\kappa')$.

In [TJ92], Talpin and Jouvelot show how to reduce both equality and subset constraints on effects to subeffecting constraints. The key point is that, in the inference system, function shape schemes have the form:

$$\forall\, \overline{v}.\, \kappa \Rightarrow \mathsf{ref}_{\rho_0}\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n\ \xrightarrow{\varphi}\ Z_0)$$

Where the latent effects of the function are specified by a single effect variable φ, which is constrained by the system κ. If the body of the function has effects F, then κ shall imply the constraint $\varphi \sqsupseteq F$. This effectively makes every function effect polymorphic. It also simplifies $\triangleright$-unification of function shapes (cf. rule [Fun], Fig. 5.1), that requires the latent effects of both shapes to be equal. (If latent effects were sets of effects as in the declarative type system, what would be a substitution for $\varphi_1 \sqcup \{write_{\rho_1}\} = \{write_{\rho_2}\} \sqcup \varphi_2$?)

5.4 Inference rules

This section provides an overview of the inference rules, and discuss a few of these rules that are of key importance. Roughly, in addition to what the declarative typing rules do, inference rules also take a subeffecting constraint system as input; and produce a substitution and a new set of subeffecting inequations. Constraint systems collect

$$\vdash_{\uparrow \mathrm{L}} : \textsc{Env} \times \mathrm{K} \times \textsc{Lval} \to \textsc{Subst} \times \textsc{Shape} \times \textsc{Effect} \times \mathrm{K}$$

[Var]
$$\frac{\Gamma(x) = \mathsf{ref}_\rho\ Z}{\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} x : \emptyset\ \&\ \mathsf{ref}_\rho\ Z\ \&\ \emptyset\ \&\ \kappa}$$

[Deref]
$$\frac{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E : \theta\ \&\ \mathsf{ptr}\ \mathsf{ref}_\rho\ Z\ \&\ F\ \&\ \kappa'}{\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} *E : \theta\ \&\ \mathsf{ref}_\rho\ Z\ \&\ F\ \&\ \kappa'}$$

[Fun]
$$\frac{\begin{array}{c}\Gamma(f) = \forall\ \overline{\rho}\overline{\zeta}\overline{\varphi}.\ \kappa_0 \Rightarrow \mathsf{ref}_{\rho_0}\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\varphi_0} Z_0) \\ \theta = \{\overline{\rho \mapsto \rho'}, \overline{\zeta \mapsto \zeta'}, \overline{\varphi \mapsto \varphi'}\} \qquad \overline{\zeta'}, \overline{\rho'}, \overline{\varphi'}\ \mathsf{fresh}\end{array}}{\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} f : \emptyset\ \&\ \mathsf{ref}_{\rho_0}\ \theta(Z_1 \times \cdots \times Z_n \xrightarrow{\varphi_0} Z_0)\ \&\ \emptyset\ \&\ \kappa \sqcup \theta\kappa_0}$$

[Index]
$$\frac{\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} L : \theta\ \&\ \mathsf{ref}_{\rho_1}\ Z_1\ \&\ F_1\ \&\ \kappa' \qquad \theta\Gamma;\kappa' \vdash_{\uparrow \mathrm{E}} E : \theta'\ \&\ Z_2\ \&\ F_2\ \&\ \kappa''}{\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} L\,[E] : \theta'\theta\ \&\ \theta'(\mathsf{ref}_{\rho_1}\ Z_1)\ \&\ \theta' F_1 \cup F_2\ \&\ \kappa''}$$

[Field]
$$\frac{\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} L : \theta\ \&\ \mathsf{ref}_{\rho_0}\ \texttt{struct}\ A\ \{\ \overline{\mathsf{ref}_{\rho_i}\ Z_i\ x_i}\ \}\ \&\ F\ \&\ \kappa'}{\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} \theta : L.x_j\ \&\ \mathsf{ref}_{\rho_j}\ Z_j\ \&\ F\ \&\ \kappa'}$$

Figure 5.3: Inference rules for lvalues.

and propagate the subeffecting constraints introduced by function call invocations. The output substitution is a local solution to the equality constraints imposed by the application of the rule. Unlike subeffecting constraint systems, these equations are not guaranteed to have a solution; and it is preferable to compute the solution incrementally, and deal with errors as early as possible.

5.4.1 Inference rules for l-values

The judgment $\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} L : \theta\ \&\ Z\ \&\ F\ \&\ \kappa'$ specifies that, given the environment Γ and the constraint system κ, the l-value L has principal shape Z, its evaluation produces effects F, and introduces typing constraints θ and κ'. Figure 5.3 shows the inference rules for l-values.

For instance, rule [Fun] instantiates the shape scheme of a function. Unlike the declarative typing rules, the inference algorithm is not allowed to *guess* which concrete regions, shapes, and effects must be used.

This choice depends on the context in which the function is used (e.g., what are its actual parameters), which in general is *yet* unknown. Note that a pointer to this function reference could be passed around and be applied far away from where the reference was obtained.

The solution is to generate a *fresh* type variable for each one of the quantified variables, representing *yet-unknown* regions, shapes and effects. A substitution is build to instantiate the shape of the function accordingly. The latent effect constraints κ_0 are also instantiated, and added to the input system κ.

5.4.2 Inference rules for expressions

The judgment $\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} E : \theta \;\&\; Z \;\&\; F \;\&\; \kappa'$ specifies that, given the environment Γ and the constraint system κ, the expression E has principal shape Z, its evaluation produces effects F, and introduces typing constraints θ and κ'. Figures 5.6, 5.7, and 5.8, show the inference rules for expressions. For layout reasons, these figures are located at the of the chapter.

For instance, rule [Int-A] is used to infer the shape of an integer arithmetic expression, where the operands may be masquerading pointers. A priori, each operand may have a different shape, and $\rhd$-unification is required to constraint them to be equal. (Because, in CIL, both operands are guaranteed to have the same type, potentially after inserting type casts, the two shapes are also guaranteed to be type-compatible, and thus $\rhd$-unification will effectively enforce both shapes to be equal, or fail.)

Rule [Cast] allows the system to track operations on memory objects through type casts. The rule computes the most general shape for the target type, and then computes a substitution using $\rhd$-unification that establishes links between the memory regions of the source and target shapes. The two shapes may not even be type compatible, e.g., when casting between struct types with different memory layout.

5.4.3 Inference rules for instructions

The judgment $\Gamma; \kappa \vdash_{\uparrow \mathrm{I}} I : \theta \;\&\; F \;\&\; \kappa' \;\&\; s$pecifies that, given the environment Γ and the constraint system κ, the instruction I may have the effects F when evaluated, and introduces typing constraints θ and κ'. Figure 5.4 shows the inference rules for instructions.

$$\vdash_{\uparrow \mathrm{I}} : \textsc{Env} \times \mathrm{K} \times \textsc{Instr} \to \textsc{Subst} \times \textsc{Shape} \times \textsc{Effect} \times \mathrm{K}$$

[Set-Exp]

$$\frac{\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} L : \theta \;\&\; \mathsf{ref}_\rho\ Z_1 \;\&\; F_1 \;\&\; \kappa' \qquad \theta\Gamma;\kappa' \vdash_{\uparrow \mathrm{E}} E : \theta' \;\&\; Z_2 \;\&\; F_2 \;\&\; \kappa'' \qquad \theta' Z_1 \rightsquigarrow Z_2 = \theta''}{\Gamma;\kappa \vdash_{\uparrow \mathrm{I}} L = E : \theta''\theta'\theta \;\&\; \theta'' Z_2 \;\&\; \theta''(\theta' F_1 \cup F_2 \cup \theta'\{write_\rho\}) \;\&\; \theta''\kappa''}$$

[Call]

$$\frac{\begin{array}{c}\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} L_f : \theta_0 \;\&\; \mathsf{ref}_{\rho_0}\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\varphi} Z_0) \;\&\; F_0 \;\&\; \kappa_0 \\ \theta_{i-1}\cdots\theta_0\Gamma;\kappa_{i-1} \vdash_{\uparrow \mathrm{E}} E_i : \theta_i \;\&\; Z_i' \;\&\; F_i \;\&\; \kappa_i \\ Z_i' \rightsquigarrow Z_i = \theta_i' \qquad F_i' = \theta_{n:i+1} F_i \qquad i \in [1,n] \\ Z_0' = \theta_{n:1} Z_0 \qquad \varphi' = \theta_{n:1}\varphi \qquad \theta' = \theta'_{n:1} \qquad \theta'' = \theta'\theta_{n:0}\end{array}}{\Gamma;\kappa \vdash_{\uparrow \mathrm{I}} L_f(E_1,\cdots,E_n) : \theta'' \;\&\; \theta' Z_0' \;\&\; \theta'(F_0' \cup \bigcup_{i\in[1,n]} F_i' \cup \varphi') \;\&\; \theta'\kappa_n}$$

[Set-Call]

$$\frac{\begin{array}{c}\Gamma;\kappa \vdash_{\uparrow \mathrm{L}} L : \theta \;\&\; \mathsf{ref}_\rho\ Z \;\&\; F_1 \;\&\; \kappa' \\ \Gamma;\kappa' \vdash_{\uparrow \mathrm{I}} L_f(E_1,\cdots,E_n) : \theta' \;\&\; Z_2 \;\&\; F_2) \;\&\; \kappa'' \\ F_1' = \theta' F_1 \qquad \rho' = \theta'\rho \qquad \theta' Z_1 \rightsquigarrow Z_2 = \theta'' \qquad \theta''' = \theta''\theta'\theta\end{array}}{\Gamma;\kappa \vdash_{\uparrow \mathrm{I}} L = L_f(E_1,\cdots,E_n) : \theta''' \;\&\; \theta'' Z_2 \;\&\; \theta''(F_1' \cup F_2 \cup \{write_{\rho'}\}) \;\&\; \theta''\kappa''}$$

Figure 5.4: Inferece rules for instructions.

5.4.4 Inference rules for statements

The judgment $\Gamma;\kappa \vdash^{\Downarrow Z}_{\uparrow \mathrm{S}} S : \theta \;\&\; F \;\&\; \kappa'$ specifies that, within a function returning values of shape Z, and given the environment Γ and the constraint system κ, the statement S may have the effects F when evaluated, and introduces typing constraints θ and κ'. Figure 5.9 shows the inference rules for statements. For layout reasons, this figure is located at the of the chapter.

5.4.5 Inference rules for globals

The judgment $\Gamma;\kappa \vdash_{\uparrow \mathrm{G}} G \;\&\; \theta \;\&\; \Gamma' \;\&\; \kappa'$ specifies that, in the environment Γ and the constraint system κ, the global G is valid, transforms the environment into Γ', and and introduces typing constraints θ and κ'. Figure 5.5 shows the inference rules for globals.

$$\vdash_{\mathrm{G}} : \textsc{Env} \times \mathrm{K} \times \textsc{Global} \to \textsc{Subst} \times \textsc{Env} \times \mathrm{K}$$

[Var-Decl]
$$\frac{\Gamma' = \Gamma; x : Z \qquad \text{shape-of}(T) = Z}{\Gamma; \kappa \vdash_{\uparrow \mathrm{G}} T\ x\texttt{;}\ \&\ \emptyset\ \&\ \Gamma'\ \&\ \kappa}$$

[Var-Def]
$$\frac{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} E : \theta\ \&\ Z\ \&\ F\ \&\ \kappa' \qquad \Gamma' = \Gamma; x : Z}{\Gamma; \kappa \vdash_{\uparrow \mathrm{G}} T\ \ x\ =\ E\texttt{;}\ \&\ \theta\ \&\ \Gamma'\ \&\ \kappa'}$$

[Fun-Def]
$$\frac{\begin{array}{c} Z_i = \text{shape-of}(T_i)\ /\ i \in [0,m] \qquad \rho_0 \cdots \rho_m \varphi_0 \text{ fresh} \\ Z_f = \mathsf{ref}_{\rho_0}\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\varphi_0} Z_0) \\ \Gamma' = \Gamma; x_1 : \mathsf{ref}_{\rho_1}\ Z_1; \cdots; x_n : \mathsf{ref}_{\rho_n}\ Z_n \\ (\Gamma'; f : \forall\ .\ Z_f; \overline{x_j : \mathsf{ref}_{\rho_j}\ Z_j;}^{\,n+1:m}); \kappa \vdash^{\Downarrow Z_0}_{\uparrow \mathrm{S}} S : \theta\ \&\ F\ \&\ \kappa' \qquad \theta' = \overline{\kappa'}\theta \\ F' = \text{Observe}_{\theta'\Gamma', \theta' Z_f}(\theta' F) \qquad \overline{v} = \text{FTV}(\theta' Z_f) \setminus (\text{FTV}(\theta'\Gamma) \cup \theta\{\rho_0\}) \\ \kappa_f = \kappa'_{\overline{v}} \sqcup \{\theta\varphi_0 \sqsupseteq F'\} \qquad \Gamma'' = \Gamma; f : \forall\ \overline{v}.\ \kappa_f \Rightarrow \theta Z_f \qquad \kappa'' = \kappa' \setminus \kappa_f \end{array}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{G}} T_0\ \ f\ (T_1\ \ x_1, \cdots, T_n\ \ x_n)\ \ \{\ \ \overline{T_j\ x_j\texttt{;}}^{\,n+1:m}\ \ S\ \ \}\ \&\ \theta\ \&\ \Gamma''\ \&\ \kappa''}$$

[Tr-Unit]
$$\frac{\Gamma_{i-1}; \kappa_0 \vdash_{\uparrow \mathrm{G}} G_i\ \&\ \theta_i\ \&\ \Gamma_i\ \&\ \kappa_i \qquad i \in [1,n]}{\Gamma_0; \kappa_0 \vdash_{\uparrow \mathrm{G}} G_1 \cdots G_n\ \&\ \theta_{n:1}\ \&\ \Gamma_n\ \&\ \kappa_n}$$

. . . . . .

$$\begin{array}{l} \text{Observe}_{\Gamma,Z}(F) = \{\varepsilon(\overline{\rho}) \in F \mid \exists \rho_i \in \text{FTV}(\Gamma) \\ \qquad\qquad \cup\, \text{FTV}(Z)\} \cup \{\varphi \in F \mid \varphi \in \text{FTV}(\Gamma) \cup \text{FTV}(Z)\} \end{array}$$

Figure 5.5: Inference rules for globals.

Rule [Var-Decl] infers the shape of a declared variable. We still assume that, as in Sect.4.5.5, CIL translation units are simplified so that each global variable is either declared or define exactly once. Variables are simply given the most-general shape possible according to their declared type. This shape typically contains variables that may be constrained later on, and therefore the final shape assigned to the variable will depend on how it is used within the translation unit.

The (fairly complex) rule [Fun-Def] infers the shape scheme of a function definition. The interesting aspect of this rule is how subeffect-

ing is handled. The rule generates a fresh effect variable φ_0 that represents the latent effects of the function. Being F_0 the effects of evaluating the body of the function, the shape scheme of the function carries the constraint $\varphi_0 \sqsupseteq F_0$. Depending on the contexts where the function may be used, φ_0 can be further constrained, but it will always have to include the effects F. The subeffecting constraints resulting from inferring the shape and effects of the function body, that is κ', are split into those that constrain variables on which the function is polymorphic, that is $\kappa_{\overline{v}}$, and those that refer to effect variables with global scope. The former are carried by the shape scheme, whereas the latter are propagated to the next global.

5.5 Limitations

Real C programs, especially in the operating systems domain, do involve very complex pointer manipulation, and exploit all the possibilities that the C standard allows (and some that are implementation specific). While a shape and effect system that can handle all these unsafe uses of C precisely and efficiently might never exist, we still need to analyze C programs. The shape-and-effect inference algorithm presented here is imperfect, but it does handle a large amount of C constructs and idioms.

For the cases that it does not handle well, the implementation of the inferrer—as part of a bug-finding tool—should be able of degrading gracefully, and continue to operate in the presence of inference-related failures without crashing. For example, in EBA, the rule [Cast] is relaxed and, even though the implementation will try to perform $\rhd$-unification, even if no [Unif-*] rule matches (cf. Fig. 5.1), a cast will always succeed. In any case, the implementation shall produce an effect-based abstraction for any program that the compiler accepts.

The reasons why $\rhd$-unification may fail include the aforementioned casts from pointers to struct members to the container struct object (cf. Sect. 4.6), and also pointer manipulations that result in cyclic shapes. For instance, in `int *b; b = *b;`, the assignment `b = *b` will introduce the following cyclic constraint $\mathsf{ptr}\ \mathsf{ref}_\rho\ Z = Z$. (Yet, this is perfectly legal C code.) The implementation will simply ignore this equality constraint. Other constructs such as *inline assembly* can also have arbitrary effects, that this system does not track.

Whenever $\rhd$-unification fails, or there exist unsupported C constructs, the produced safe-and-effect abstraction could be unsound. Unsoundness means that the system can miss aliasing relationships (if $\rhd$-unification fails), or effects (e.g., for inline assembly). This can lead to both false negatives (i.e., bugs are missed) and false positives (i.e., non-bugs are reported). For instance, the system may infer that, in $S_1; S_2$, the statement S_1 has effects $\{lock_{\rho_1}\}$, and S_2 has effects $\{lock_{\rho_2}\}$, whereas in reality ρ_1 and ρ_2 point to the same object, and therefore a potential double-lock would not be detected. Similarly, for $S_1; S_2; S_3$, where the system infers that S_1 has effects $\{lock_{\rho_1}\}$, S_2 has effects $\{unlock_{\rho_2}\}$, and S_3 has effects $\{lock_{\rho_1}\}$; if ρ_1 and ρ_2 point to the same object, the bug-finder may report a non-existent double-lock.

A bug-finding tool built on top of this shape-and-effect system needs to be aware of the above limitations. EBA, for instance, employs heuristics to remove false positives caused by aliasing imprecisions (cf. Chapter 6). A more sophisticated implementation could keep track of which memory regions are involved in a failed $\rhd$-unification attempt, and down prioritize bug alarms concerning those regions. Some false negatives could be avoided by combining multiple levels of abstractions; e.g., by performing a more precise alias analysis on specific parts of the code.

5.6 Principality

I have proven no theorems about this shape and effect system. In particular, I have not proven principality of the inference system. Yet, the original work of Talpin-Jouvelot [TJ92] does guarantee principality, and I have followed their method closely. I have no reason to believe that the same does not hold here—at least in the absence of unsafe casts between different record types.

In this system, the notion of principal type should correspond to the notion of most-general shape. Thus, the shape inferred for any expression E should be the most general possible—with respect to $\rhd$, that is $\succ$-compatible with the type of E. Note that without requiring compatibility with the type of the expression, there may not be a principal shape, due to struct-to-struct conversions.

In any case, let us assume that the inference algorithm presented here does not infer the most general shape possible. The side-effect would be that $\rhd$-unification could fail more often. Unification failures are handled gracefully, but they introduce more aliasing imprecisions into the

inferred abstraction. Hence, this would lead to extra false positives and negatives (cf. Sect. 5.5). But, since we expect that any implementation will (at least) have mechanisms to remove false positives, this mostly translates in less bugs being found. False negatives are inherent to the design of lightweight bug-finding techniques.

While I have no proofs to show, I do have a proof-of-concept implementation of this shape-and-effect inference system, and a bug-finding tool (EBA) built on top of it (cf. Chapter 6). The good results obtained during the evaluation of this tool (cf. Chapter 7) constitute empirical evidence that the inference system does allow for efficient and effective inter-procedural reasoning about resource manipulation.

$$\vdash_{\uparrow \mathrm{E}} : \textsc{Env} \times \mathrm{K} \times \textsc{Exp} \to \textsc{Subst} \times \textsc{Shape} \times \textsc{Effect} \times \mathrm{K}$$

[Const-Bot]
$$\frac{\text{typeof}(c) \in \{\texttt{float}, \texttt{double}\}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} c : \emptyset \;\&\; \bot \;\&\; \emptyset \;\&\; \kappa}$$

[Const-Str]
$$\frac{\text{typeof}(str) = \texttt{char}\star \qquad \rho \text{ fresh}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} str : \emptyset \;\&\; \text{ptr ref}_\rho \; \bot \;\&\; \emptyset \;\&\; \kappa}$$

[Const-Int]
$$\frac{\text{typeof}(i) \in \{\texttt{char}, \texttt{int}, \texttt{short}, \texttt{long}, \texttt{long long}\} \qquad \zeta \text{ fresh}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} i : \emptyset \;\&\; \zeta \;\&\; \emptyset \;\&\; \kappa}$$

[Lval]
$$\frac{\Gamma; \kappa \vdash_{\uparrow \mathrm{L}} L : \theta \;\&\; \text{ref}_\rho \; Z \;\&\; F \;\&\; \kappa'}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} L : \theta \;\&\; Z \;\&\; F \cup \{read_\rho\} \;\&\; \kappa'}$$

[Addr]
$$\frac{\Gamma; \kappa \vdash_{\uparrow \mathrm{L}} L : \theta \;\&\; \text{ref}_\rho \; Z \;\&\; F \;\&\; \kappa'}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} \star L : \theta \;\&\; \text{ptr ref}_\rho \; Z \;\&\; F \;\&\; \kappa'}$$

[Question]
$$\frac{\begin{array}{c}\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} E_1 : \theta_1 \;\&\; Z_1 \;\&\; F_1 \;\&\; \kappa_1 \qquad \theta_1\Gamma; \kappa_1 \vdash_{\uparrow \mathrm{E}} E_2 : \theta_2 \;\&\; Z_2 \;\&\; F_2 \;\&\; \kappa_2 \\ \theta_2\theta_1\Gamma; \kappa_2 \vdash_{\uparrow \mathrm{E}} E_3 : \theta_3 \;\&\; Z_3 \;\&\; F_3 \;\&\; \kappa' \qquad \theta_3 Z_2 \rightsquigarrow Z_3 = \theta_4\end{array}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} (E_1) \;?\; E_2 \;:\; E_3 : \theta_4\theta_3\theta_2\theta_1 \;\&\; \theta_4 Z_3 \;\&\; \theta_4(\theta_3(\theta_2 F_1 \cup F_2) \cup F_3) \;\&\; \theta_4\kappa'}$$

[Cast]
$$\frac{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} E : \theta \;\&\; Z \;\&\; F \;\&\; \kappa' \qquad Z' = \text{shape-of}(T) \qquad Z \rightsquigarrow Z' = \theta'}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} (T)\, E : \theta'\theta \;\&\; \theta' Z' \;\&\; \theta' F \;\&\; \theta' \kappa'}$$

Figure 5.6: Inference rules for non-aritmetic expressions.

$$\vdash_{\uparrow \mathrm{E}} : \textsc{Env} \times \mathrm{K} \times \textsc{Exp} \to \textsc{Subst} \times \textsc{Shape} \times \textsc{Effect} \times \mathrm{K}$$

[Sizeof-T]
$$\frac{T \neq T'\,[E] \qquad \zeta \text{ fresh}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} \texttt{sizeof}(T) : \emptyset \;\&\; \zeta \;\&\; \emptyset \;\&\; \kappa}$$

[Sizeof-A]
$$\frac{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} E : \theta \;\&\; Z \;\&\; F \;\&\; \kappa' \qquad \zeta \text{ fresh}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} \texttt{sizeof}(T\,[E]) : \theta \;\&\; \zeta \;\&\; F \;\&\; \kappa'}$$

[Sizeof-E]
$$\frac{\zeta \text{ fresh}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} \texttt{sizeof}(E) : \emptyset \;\&\; \zeta \;\&\; \emptyset \;\&\; \kappa}$$

[Alignof-T]
$$\frac{\zeta \text{ fresh}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} \texttt{alignof}(T) : \emptyset \;\&\; \zeta \;\&\; \emptyset \;\&\; \kappa}$$

[Alignof-E]
$$\frac{\zeta \text{ fresh}}{\Gamma; \kappa \vdash_{\uparrow \mathrm{E}} \texttt{alignof}(E) : \emptyset \;\&\; \zeta \;\&\; \emptyset \;\&\; \kappa}$$

Figure 5.7: Inference rules for `sizeof`() and `alignof`() expressions.

$$\vdash_{\uparrow \mathrm{E}} : \textsc{Env} \times \mathrm{K} \times \textsc{Exp} \to \textsc{Subst} \times \textsc{Shape} \times \textsc{Effect} \times \mathrm{K}$$

[Neg]
$$\frac{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E : \theta \;\&\; Z \;\&\; F \;\&\; \kappa' \qquad \ominus \in \{-, \sim\}}{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} \ominus E : \theta \;\&\; Z \;\&\; F \;\&\; \kappa'}$$

[Not]
$$\frac{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E : \theta \;\&\; Z \;\&\; F \;\&\; \kappa'}{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} \,!E : \theta \;\&\; \bot \;\&\; F \;\&\; \kappa'}$$

[Int-A]
$$\frac{\begin{array}{c}\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 : \theta \;\&\; Z_1 \;\&\; F_1 \;\&\; \kappa' \qquad \theta\Gamma;\kappa' \vdash_{\uparrow \mathrm{E}} E_2 : \theta' \;\&\; Z_2 \;\&\; F_2 \;\&\; \kappa'' \\ \theta' Z_1 \rightsquigarrow Z_2 = \theta'' \qquad \oplus \in \{+, -, \star, /, \%, \&, \hat{}, |, \ll, \gg\}\end{array}}{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 \oplus E_2 : \theta''\theta'\theta \;\&\; \theta'' Z_2 \;\&\; \theta''(\theta' F_1 \cup F_2) \;\&\; \theta''\kappa''}$$

[Plus-PI]
$$\frac{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 : \theta \;\&\; \mathsf{ptr}\ \mathsf{ref}_\rho\ Z_1 \;\&\; F_1 \;\&\; \kappa' \qquad \theta\Gamma;\kappa' \vdash_{\uparrow \mathrm{E}} E_2 : \theta' \;\&\; Z_2 \;\&\; F_2 \;\&\; \kappa''}{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 + E_2 : \theta'\theta \;\&\; \theta'(\mathsf{ptr}\ \mathsf{ref}_\rho\ Z_1) \;\&\; \theta' F_1 \cup F_2 \;\&\; \kappa''}$$

[Minus-PP]
$$\frac{\begin{array}{c}\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 : \theta \;\&\; \mathsf{ptr}\ \mathsf{ref}_{\rho_1}\ Z_1 \;\&\; F_1 \;\&\; \kappa' \\ \theta\Gamma;\kappa' \vdash_{\uparrow \mathrm{E}} E_2 : \theta' \;\&\; \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ Z_2 \;\&\; F_2 \;\&\; \kappa'' \\ \mathsf{ptr}\ \mathsf{ref}_{\rho_1}\ Z_1 \rightsquigarrow \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ Z_2 = \theta''\end{array}}{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 - E_2 : \theta''\theta'\theta \;\&\; \bot \;\&\; \theta''(\theta' F_1 \cup F_2) \;\&\; \theta''\kappa''}$$

[Bool-A]
$$\frac{\begin{array}{c}\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 : \theta \;\&\; Z_1 \;\&\; F_1 \;\&\; \kappa' \\ \theta\Gamma;\kappa' \vdash_{\uparrow \mathrm{E}} E_2 : \theta' \;\&\; Z_2 \;\&\; F_2 \;\&\; \kappa'' \qquad \odot \in \{\&\&, ||\}\end{array}}{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 \odot E_2 : \theta'\theta \;\&\; \bot \;\&\; \theta' F_1 \cup F_2 \;\&\; \kappa''}$$

[Cmp]
$$\frac{\begin{array}{c}\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 : \theta \;\&\; Z_1 \;\&\; F_1 \;\&\; \kappa' \\ \theta\Gamma;\kappa' \vdash_{\uparrow \mathrm{E}} E_2 : \theta' \;\&\; Z_2 \;\&\; F_2 \;\&\; \kappa'' \qquad \trianglelefteq \in \{<, >, <=, >=, ==, !=\}\end{array}}{\Gamma;\kappa \vdash_{\uparrow \mathrm{E}} E_1 \trianglelefteq E_2 : \theta'\theta \;\&\; \bot \;\&\; \theta' F_1 \cup F_2 \;\&\; \kappa''}$$

Figure 5.8: Inference rules for arithmetic expressions.

$$\vdash_{\uparrow S} : \textsc{Env} \times \mathrm{K} \times \textsc{Shape} \times \textsc{Stmt} \to \textsc{Subst} \times \textsc{Effect} \times \mathrm{K}$$

[Instr]
$$\frac{\Gamma;\kappa \vdash_{\uparrow I} I : \theta \;\&\; Z' \;\&\; F \;\&\; \kappa'}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} I\texttt{;} : \theta \;\&\; F \;\&\; \kappa'}$$

[Return]
$$\frac{}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} \texttt{return;} : \emptyset \;\&\; \emptyset \;\&\; \kappa}$$

[Return-E]
$$\frac{\Gamma;\kappa \vdash_{\uparrow E} E : \theta \;\&\; Z' \;\&\; F \;\&\; \kappa' \qquad Z' \rightsquigarrow Z = \theta'}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} \texttt{return}\ E\texttt{;} : \theta'\theta \;\&\; \theta' F \;\&\; \theta'\kappa'}$$

[Label]
$$\frac{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} S : \theta \;\&\; F \;\&\; \kappa'}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} l\texttt{:}\quad S\texttt{;} : \theta \;\&\; F \;\&\; \kappa'}$$

[Goto]
$$\frac{}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} \texttt{goto}\ l\texttt{;} : \emptyset \;\&\; \emptyset \;\&\; \kappa}$$

[Break]
$$\frac{}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} \texttt{break;} : \emptyset \;\&\; \emptyset \;\&\; \kappa}$$

[Continue]
$$\frac{}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} \texttt{continue;} : \emptyset \;\&\; \emptyset \;\&\; \kappa}$$

[If]
$$\frac{\begin{array}{c}\Gamma;\kappa \vdash_{\uparrow E} E : \theta_0 \;\&\; Z_0 \;\&\; F_0 \;\&\; \kappa_0 \\ \theta_0\Gamma;\kappa_0 \vdash_{\uparrow S}^{\Downarrow Z} S_1 : \theta_1 \;\&\; F_1 \;\&\; \kappa_1 \qquad \theta_1\theta_0\Gamma;\kappa_1 \vdash_{\uparrow S}^{\Downarrow Z} S_2 : \theta_2 \;\&\; F_2 \;\&\; \kappa_2\end{array}}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} \texttt{if}\ \ (E)\ \ S_1\ \ S_2 : \theta_2\theta_1\theta_0 \;\&\; \theta_2\theta_1 F_0 \cup \theta_2 F_1 \cup F_2 \;\&\; \kappa_2}$$

[Switch]
$$\frac{\Gamma;\kappa \vdash_{\uparrow E} E : \theta_0 \;\&\; Z_0 \;\&\; F_0 \;\&\; \kappa_0 \qquad \theta_{i-1:0}\Gamma;\kappa_{i-1} \vdash_{\uparrow S}^{\Downarrow Z} S_i : \theta_i \;\&\; F_i \;\&\; \kappa_i \;/\; i \in [1,n]}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} \texttt{switch}\ \ (E)\ \ \{\ \ S_1\ \ \cdots\ \ S_n\ \ \} : \theta_{n:0} \;\&\; \theta_{n:1} F_0 \cup (\bigcup_{i\in[1,n]} \theta_{n:i+1} F_i) \;\&\; \kappa_n}$$

[Loop]
$$\frac{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} S : \theta \;\&\; F \;\&\; \kappa'}{\Gamma;\kappa \vdash_{\uparrow S}^{\Downarrow Z} \texttt{while}\ \ (1)\ \ S : \theta \;\&\; F \;\&\; \kappa'}$$

[Block]
$$\frac{\Gamma;\kappa_{i-1} \vdash_{\uparrow S}^{\Downarrow Z} S_i : \theta_i \;\&\; F_i \;\&\; \kappa_i \;/\; i \in [1,n]}{\Gamma;\kappa_0 \vdash_{\uparrow S}^{\Downarrow Z} \{S_1 \cdots S_n\} : \theta_{n:1} \;\&\; \bigcup_{i\in[1,n]} \theta_{n:i+1} F_i \;\&\; \kappa_n}$$

Figure 5.9: Inference rules for statements.

Chapter 6

Effective Bug Finding with EBA

This chapter presents the bug-finding technique that makes the ultimate contribution of this PhD work. I have implemented it in the EBA tool. This technique has been sketched earlier in Sect. 1.4. This chapter also discusses key aspects of its implementation. EBA is implemented in OCaml and built on top of the CIL [NMRW02] front-end infrastructure. Its source code is publicly available under an open-source license.[1]

Figure 6.1 shows the analysis pipeline implemented by EBA. At present, and purely for simplicity, EBA analyzes individual C files in isolation. (Multiple C files can be merged into a single file to be analyzed.) The *front-end* (Sect. 6.1) is responsible for parsing the C file, and generating a CIL *abstract syntax tree* (AST). After this, the *inferrer* (Sect. 6.2) takes this AST and decorates it with shape-and-effect information, producing an effect-annotated *control-flow graph* (*F*-CFG). Then, the *model-checker* (Sect. 6.3) traverses this *F*-CFG in search of bugs. Finally, a *bug filter* (Sect. 6.4) removes duplicates and false positives.

6.1 Front-End: from C to CIL

EBA uses the CIL front-end to parse the input C file, and transform it into an analysis-friendly intermediate representation (IR). The input C file is generated by the build system (Sect. 6.1.1), which already runs the C preprocessor if necessary. After the C-to-CIL conversion, the constructed AST is *post-processed* (Sect. 6.1.2)—right now this is just to eliminate dead code. The front-end is a quite important part of a tool. Using CIL was

[1] http://www.iagoabal.eu/eba/

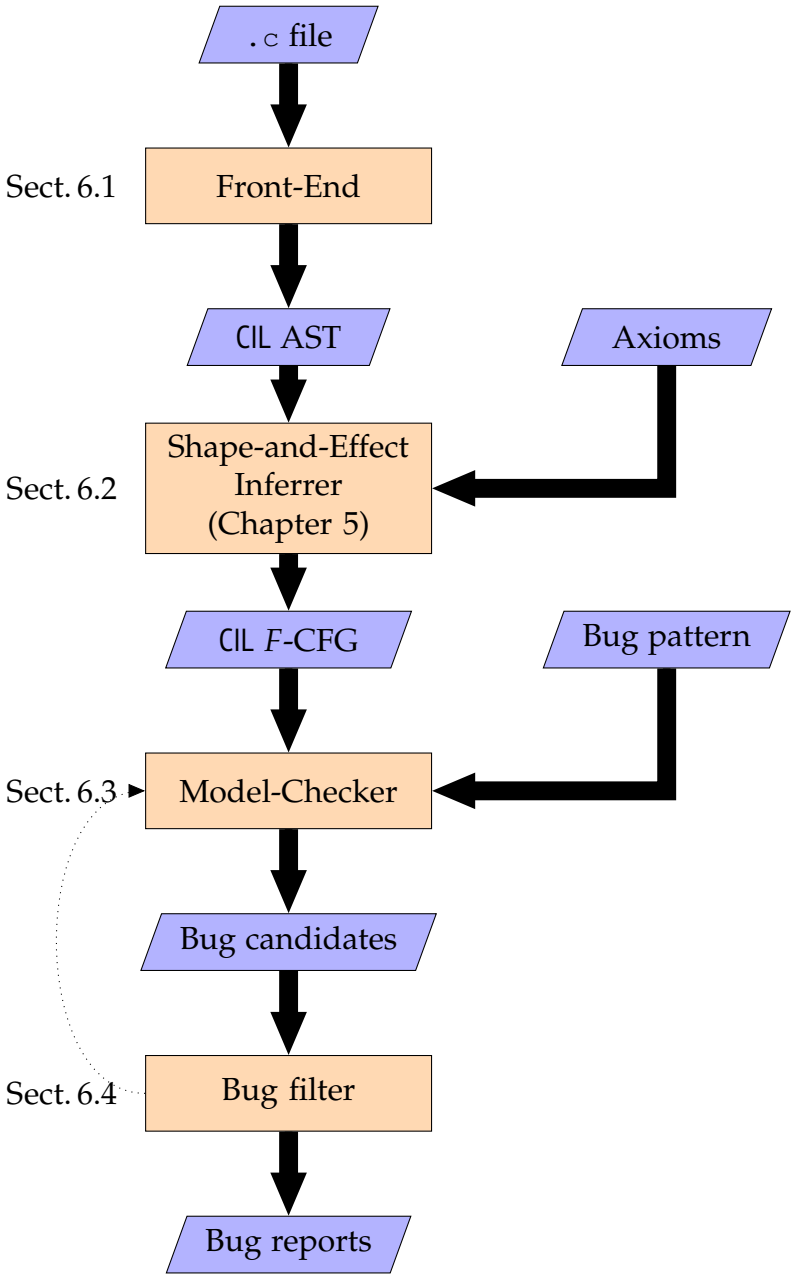


Figure 6.1: The EBA analysis pipeline: Analyzing a single C file. The blue trapeziums represent analysis data, whereas the orange rectangles represented analysis processes.

convenient for a proof-of-concept prototype, but there exist interesting alternatives to consider (Sect. 6.1.3).

6.1.1 Integration with the build system

A typical problem when analyzing a software project, is to know which files need to be analyzed in the first place. The `.c` source files will not contain pure C code, but code with preprocessor directives, macro definitions, and macro uses. These files need to be preprocessed before being suitable for analysis (more on this later), and the way of doing this is only known to the build system. For example, in Kbuild systems like Linux, the result of preprocessing depends on which configuration options are set. It is possible to obtain these files in a way that is agnostic of the build system, by *capturing* invocations to the preprocessor or compiler, and extracting the relevant command-line arguments. This can be done, for instance, by using a debugging tool like strace, which can monitor all *exec* syscalls made by a running build command.

Fortunately, no sophisticated mechanism is required to run EBA on Linux source code. The Linux build system already provides a source code checking facility. It allows to specify a command to *check* the C sources, through the use of the `CHECK` environment variable. If variable `C` is defined, then every `.c` file to be compiled will be checked with the command specified in `$CHECK`, receiving the same command-line arguments as the compiler. EBA is distributed with a simple script, called *eba-gcc*, that provides a GCC-like wrapper, which preprocesses the input file and then runs EBA on the result. One can run EBA on Linux by simply typing `make C=1 CHECK="eba-gcc"`. This will run EBA on the files to be (re)compiled; to force it to run on all the files (regardless of whether they need to be recompiled), set `C=2`.

6.1.2 Reducing the input code

CIL takes preprocessed files as input. Preprocessed files tend to be large (in the order of tens of KLOC) and contain thousands of function definitions. Most of these function definitions come from included headers, most of them unused in the input `.c` file. (Some header files contain many utility functions, of which each particular input file may only use a few.) EBA uses CIL dead code elimination facilities to remove all the unused function definitions (and other unused declarations as well) to

decrease the input size. Experiments have shown that, in average, only 2% of the function definitions are used.

6.1.3 Alternative front-ends

I have chosen CIL because it is a native OCaml library, and its intermediate representation (IR) is stable and, in fact, a subset of C. CIL IR is amenable to static analysis and, by being a subset of C, it allows EBA to refer to code that the programmer can recognize as his own. On the other hand, GCC and LLVM IRs are lower-level representations, designed for efficient compilation. Also, they are considered part of the compiler internals, and therefore can easily be subject to backwards incompatible changes.

After I had started developing EBA, Facebook released the source code of INFER. INFER uses Clang to parse the C files, exports the constructed AST using a Clang plugin, and finally imports and rebuilds the AST in OCaml. Then, they perform a transformation of the C AST into a CIL-like IR. This saves the INFER team from maintaining an industrial-strength C parser. I would like to consider this approach in the future, perhaps even reusing the INFER front-end. Although the performance penalty of importing the AST that is built by Clang into OCaml would need to be quantified.

As discussed earlier, the problem with these compiler front-ends is that they only handle pure C code. This requires integrating the bug finding tool with the build system which, in some cases, may not be trivial to do. Some software projects, like Linux, rely heavily on macros to define higher-level constructs (e.g., `container_of`[2]), or even to annotate the code (e.g., `__acquires`[3]). After preprocessing, the meaning of these constructs and annotations can be lost. Given enough resources, I would have considered writing a *fuzzy* front-end, similar to that of Coccinelle or CppCheck, capable of parsing C with preprocessor. This would also make it easier to perform configuration-sensitive analysis (cf. Sect. 1.1).

6.2 Shape-and-effect inferrer

The inferrer implements the shape-and-effect inference algorithm of Chapter 5. Given a CIL AST, it infers the memory shapes and aliasing relationships of all program variables, and the effects for all statements.

[2] `http://lxr.free-electrons.com/ident?i=container_of`
[3] `http://lxr.free-electrons.com/ident?i=__acquires`

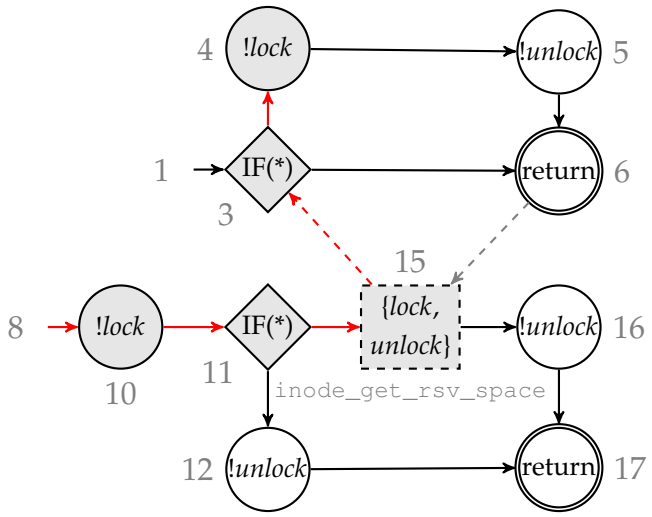


Figure 6.2: An illustration of our bug-finding technique for the double-lock bug in Figure 1.2. The figure shows the associated CFG annotated with lock and unlock effects. The numbers next to the CFG nodes show corresponding line numbers. The gray nodes visualize the (red) path, via the function call in line 15, to the double-lock (in line 4).

Crucially, each function is assigned a shape-and-effect signature, which establishes aliasing relationships between inputs and outputs, and provides a flow-insensitive summary of its observable behavior. This information is superimposed on the CFG, obtaining the so-called effect-based CFG, or *F*-CFG, of the program. Figure 6.2 shows an example *F*-CFG (basically the same as that of Fig. 1.3, repeated here for convenience).

We begin with a standard CFG, where nodes represent program locations, and edges specify the control-flow. We distinguish branching decisions (diamond nodes), atomic operations (circles), function calls (dotted squares), and *return* statements (double-circles). A *F*-CFG is an effect-abstraction of a program obtained from the standard CFG by annotating variables with their memory shapes, and nodes with the effects inferred for the corresponding locations. Function call nodes hold a flow-insensitive over-approximation of the callee's behavior. For instance, in Fig. 6.2, the call to `inode_get_rsv_space` is summarized as $\{lock, unlock\}$, indicating that this function call may both acquire and release `inode->lock`.

$$\vdash_S \subseteq \textsc{Env} \times \textsc{Shape} \times \textsc{Stmt} \times \textsc{Effect}$$

$$[\textsc{If}]\ \frac{\Gamma \vdash_E E : Z_0 \ \&\ F_0 \qquad \Gamma \vdash_S^{\Downarrow Z} S_1 \ \&\ F_1 \qquad \Gamma \vdash_S^{\Downarrow Z} S_2 \ \&\ F_2}{\Gamma \vdash_S^{\Downarrow Z} \texttt{if (}E\texttt{)}\ S_1\ S_2 \ \&\ F_0 \cup \mathbf{may}F_1 \cup \mathbf{may}F_2}$$

$$[\textsc{Switch}]\ \frac{\Gamma \vdash_E E : Z_0 \ \&\ F_0 \qquad \Gamma \vdash_S^{\Downarrow Z} S_i \ \&\ F_i \ /\ i \in [1,n]}{\Gamma \vdash_S^{\Downarrow Z} \texttt{switch (}E\texttt{) \{}\ S_1 \cdots S_n\ \texttt{\}} \ \&\ F_0 \cup \left(\bigcup_{i\in[1,n]} \mathbf{may}F_i\right)}$$

$$[\textsc{Block}]\ \frac{\Gamma \vdash_S^{\Downarrow Z} S_i \ \&\ F_i \ /\ i \in [1,n]}{\Gamma \vdash_S^{\Downarrow Z} \{S_1 \cdots S_n\} \ \&\ F_1 \cup \left(\bigcup_{i\in[\mathbf{2},n]} \mathbf{may}F_i\right)}$$

Figure 6.3: Typing of statements with *must*-effects. Only the rules that change are shown, and those changes are marked using a bold blue font.

6.2.1 May and must effects

The shape-and-effect system of Chapter 4 infers only *may*-effects, whereas EBA also considers *must*-effects. The may-effects of a function describe all the effects that *may* result from applying such function, but not all applications of the function necessarily have all those effects. For instance, if a function has a may-effect $lock_\rho$, it may or may not acquire a lock on ρ, depending on the flow of control.

The must-effects of a function describe the effects that any invocation of the function is guaranteed to have. For instance, function `spin_lock` will only return after acquiring a lock. Must-effects are prefixed by a bang (!), as in $!lock_\rho$. Must-effects are useful to *mark* the basic operations that perform such effects (see Sect. 6.2.2). The bug-filter (cf. Sect. 6.4) can rely on must-effects to rank the bug reports, and decide whether inlining is necessary (more on this later).

However, the inferrer is not able to infer must-effects—that would require a flow-sensitive analysis. When the inferrer cannot guarantee that must-effects hold for a statement, it simply *downgrades* them to may-effects. Figure 6.3 shows how the typing rules are changed to deal with must-effects. The operator *may* turns must-effects into may-effects, e.g., $may(\{!lock_{\rho_2}\}) = \{lock_{\rho_2}\}$ (so it just "drops the bang"). Note that in a

sequence of statements $S_1 ; S_2$, S_2 may never execute if S_1 either diverges or aborts the execution (rule [BLOCK]).

6.2.2 Axioms

The shape-and-effect inferrer knows about a few built-in effects that are inherent to the C language, such as reads and writes to l-values. Yet, bug checkers are often more interested in tracking effects associated with specialized APIs. For EBA to track these effects, it is necessary to *mark* the elementary API functions that are the root of those effects. (Functions that are built on top of these elementary functions do not need to be annotated, if their source code is available to EBA.) EBA offers two mechanisms for doing this.

Full axioms

A *full axiom* is simply a shape scheme that is associated with a function name, and added to the global typing environment as an axiom. The axiom fully specifies the shape and effects of the function, and even if a definition of the function exists, it is ignored. Full axioms can be burdensome to specify, and thus should be used to specify simple function signatures, mainly when the code is not available. For instance, in EBA, the libc function `free` is axiomatized as follows:

$$\texttt{free} : \forall\ \rho_1\rho_2\zeta.\ \mathsf{ref}_{\rho_1}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ \zeta \xrightarrow{\{!read_{\rho_1}, !free_{\rho_2}\}} \bot \tag{6.1}$$

This axiom specifies that `free` takes a pointer to an arbitrary memory location (ρ_2), containing an object of an arbitrary shape (ζ), and it has the effect of *freeing* that chunk of memory from the heap.

Partial axioms

A *partial axiom* allows to refine the shape scheme of a function, that has been previously inferred. Usually, the refinement consists in extending the set of latent effects. Partial axioms are preferred when functions manipulate struct types, or when the source code of the function is available. (It is not a good practice to specify a complete struct shape in an axiom, since the declaration may change, or even be configuration-dependent.) For instance, Linux spin locks are mainly manipulated

through the use of the `spin_lock` and `spin_unlock` functions, which have the following prototype: `void f(spinlock_t *lock)`. EBA contains the following partial axioms to track operations on spin locks:[4]

$$\texttt{spin\_lock} : \mathsf{ref}_{\rho_1}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ Z \xrightarrow{+!lock_{\rho_2}} \bot \tag{6.2}$$

$$\texttt{spin\_unlock} : \mathsf{ref}_{\rho_1}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ Z \xrightarrow{+!unlock_{\rho_2}} \bot \tag{6.3}$$

These two partial axioms shall be understood as patterns; where ρ_1, ρ_2 and Z are meta-variables. The effects prefixed by a plus ($+$) symbol on the function arrow specify additions to the latent effects of the function. For instance, when EBA finds the definition of `spin_lock`, it will infer a shape for it, and then it will match the axiom pattern against the inferred shape, and add $lock_{\rho_2}$ to the latent effects of `spin_lock`. Note that function `spin_lock` will have a $read_{\rho_1}$ effect as well, but this effect will already be inferred by EBA—so it is omitted in the partial axiom.

Full axioms may be preferred over partial axioms because they offer slightly better performance. But, what is more important, they can be used to circumvent the inference system, since the axiom will be accepted blindly without even considering the definition of the axiomatized function.

6.2.3 Mutually recursive struct types

The shape of a struct type is obtained by recursively computing the shape of each of its members (cf. Fig. 5.2). For mutually recursive structs, such as `A` and `B` in Fig. 6.4, this would lead to infinite shapes. In practice, EBA proceeds by first constructing a struct dependency graph, and then computing the strongly connected components (SCCs). The SCCs constitute groups of mutually recursive structs. The shape of an SCC is obtained as a fixpoint of the shapes of the constituent structs. Thus, for the example of Fig. 6.4, the shape of these two structs is computed as the least fixpoint of the following equations:

$$Z_A = \texttt{struct A \{}\ \zeta_1\ \texttt{x;}\ \ \mathsf{ptr}\ \mathsf{ref}_{\rho_1}\ Z_B\ \texttt{b; \}} \tag{6.4}$$

$$Z_B = \texttt{struct B \{}\ \zeta_2\ \texttt{y;}\ \ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ Z_A\ \texttt{a; \}} \tag{6.5}$$

The result is (indeed) a cyclic shape. Cyclic struct shapes are allowed, and correctly handled, by EBA.

[4] A spin lock is, in fact, a wrapper around a lower-level *raw lock*, so EBA tracks the corresponding operations `_raw_spin_lock` and `_raw_spin_unlock`.

```
1  struct B;
2
3  struct A {
4      int x;
5      struct B *b;
6  }
7
8  struct B {
9      int y;
10     struct A *a;
11 }
```

Figure 6.4: Two mutually recursive struct declarations in C.

6.3 Model-checker

Matching execution patterns representing bugs can be reduced to the standard CTL model-checking problem over the *F*-CFG graph. A *F*-CFG is interpreted as a transition system where program statements act as states, and effects act as propositions. For instance, a proposition $lock_\rho$ holds in a state (statement) S_i iff the effects of S_i include $lock_\rho$. For instance, the absence of double-locks could be expressed as a safety property: [5]

$$\mathsf{AG}\ (lock_\rho\ \Rightarrow\ \mathsf{AX}\ (unlock_\rho\ \mathsf{AR}\ \neg lock_\rho)) \tag{6.6}$$

Essentially, the formula says that the acquisition of a lock ρ at some point in an execution path, implies that no other acquisition is performed until the lock is released.

The *F*-CFG of each function is analyzed in isolation. Function calls are treated as black-boxes, relying on their effect-summaries. A *match* (a counterexample of the safety property) is a bug candidate represented by an *error trace* (e.g., the red path in Fig. 6.2). If no counterexample is found, we may regard the function as "correct" (modulo subtle tricky uses of pointers and type-casts). Bugs may be missed due to unsoundness of the inference system (cf. sections 4.6 and 5.5). This is part of a necessary trade-off.

[5] For presentation purposes, these formulas are simplified. The real formulas are more cumbersome due to the need to rule out false positives. Note that aliasing information can be imprecise and ρ may not denote a unique runtime lock object. I encourage the interested reader to look at the actual source code.

```
1  let find_match guard target tree0 =
2          let rec traverse path tree =
3              match tree with
4              | Nil ->
5                  backtrack
6              | Assume(e,tree′) ->
7                  let path′ = add e to path in
8                  traverse path′ tree′
9              | Branch(tree1,tree2) ->
10                 traverse path tree1
11                 traverse path tree2
12             | Step(step,tree′) ->
13                 if target(step)
14                 then report match;
15                 if guard(step)
16                 then traverse path tree′
17                 else backtrack
18         in
19         traverse empty-path tree0
```

Figure 6.5: Reachability checking on the path-tree of an *F*-CFG.

6.3.1 Reachability checking

EBA implements a simple reachability checker, rather than a fully-fledged CTL model checker—the full generality was never needed. A reachability checker is easier to implement, and it naturally provides an error trace when a bug is found. Figure 6.5 shows a simplified version of the algorithm that EBA implements, in OCaml-based pseudo-code. This algorithm is capable of finding models for CTL-formulas of the form *guard* EU *target*. That is, it finds paths that reach a state where *target* holds, and *guard* holds in all previous states.

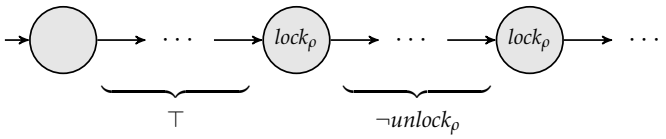
EBA separates the traversal of the CFG from the reachability check. From the *F*-CFG, EBA produces a lazy *enumerator* for all the paths in the CFG. This enumerator, conceptually a path-tree, is traversed by the reachability checker in a depth-first manner. The tree has four kinds of nodes: *Nil* marks the end of a path (e.g., after a `return`); *Assume* characterizes the path with a Boolean condition; *Branch* provides alternative flows of execution; and *Step* specifies a basic program step (i.e., a group of instructions, a branching condition, or a `return` statement).

6.3.2 Bug checkers

The first step towards creating a bug checker in EBA is to characterize the bug pattern using an existential CTL-formula—with effects as atomic propositions. The formulas must describe incorrect execution paths. For instance, an execution containing a double-lock bug can be matched using the following CTL formula (the dual of formula 6.6):

$$\top \ \mathsf{EU} \ (\mathit{lock}_\rho \ \wedge \ \mathsf{EX} \ (\neg \mathit{unlock}_\rho \ \mathsf{EU} \ \mathit{lock}_\rho)) \tag{6.7}$$

In this formula, the region ρ works as a meta variable specifying that we are interested in finding a second lock on the *same* memory object, rather than two unrelated lock acquisitions. This formula reveals buggy execution paths of the form (cf. Fig. 6.2):



The next step is to decompose this existential CTL-formula into subformulas (queries) of the form *guard* EU *target*, that can be handed to the reachability checker. The bug checker is then a small OCaml script that glues these reachability queries to find a model for the bug CTL-pattern. Continuing with our example, formula 6.7 can be decomposed into two queries: $\top$ EU lock_ρ and $\neg \mathit{unlock}_\rho$ EU lock_ρ.

6.3.3 Path pruning

Path explosion is a problem even when analyzing a single function. A single loop can add an undetermined number of paths. Besides that, a sequence of n `if` statements introduces up to 2^n possible execution paths. EBA deals with this issue in two different ways.

Bounded search. The traversal of the *F*-CFG is bounded by two parameters. The first parameter sets a bound to the number of times that an edge is taken within a path, preventing looping infinitely. The second parameter sets a bound to the number of times that a path is forked due to branching constructs, after which the search will explore only one of the branching alternatives, preventing path explosion. The default bounds are generous and allow exploring about a thousand paths,

which is enough for most functions, but some large functions will inevitably reach these bounds. (Default bounds can be overridden by using the appropriate command-line options.)

Fact tracking. Not all *potential* paths are, in fact, *feasible*. Branching decisions, and changes to the program state, will exclude some paths. For instance, let us consider the following program:

```
1 if (e) S1
2 S2
3 if (e) S3
```

If e has evaluated to v in line 1, and neither S_1 nor S_2 alter any of the memory locations read by e, then e is guaranteed to evaluate to v in line 3 as well. (Recall that CIL expressions are side-effect free.) The model-checker tracks branching decisions and other simple facts that can be derived syntactically from the code, and uses them to prune the search. Equally important, this removes false positives. By default, EBA uses a small set of rules to make basic deductions from the set of known facts. This has worked quite well for Linux, but other applications may benefit from specialized solvers.

6.4 Bug filter

Given a bug pattern and an F-CFG, the model-checker produces a set of *bug candidates*. The *bug filter* is responsible for removing duplicates and false positives, and deriving a set of *bug reports*.

6.4.1 Removal of duplicates

Often times, the same bug can be reached through multiple paths. Multiple occurrences of the same bug are grouped together, and only one of them (if any) is reported. EBA heuristically tries to pick the *simplest* bug trace of each group. Short bug traces that reach the bug in the smallest number of steps are preferred. For equally short traces, EBA picks the trace with the longest prefix of *false* decisions. The *false* valuation of a branching decision corresponds with taking the *else*, or *fall-through*, alternative of an `if` statement. This last heuristic is less obvious, but let us consider the following code containing a double-lock bug:

```
1 if (e1) S1;
2 if (e2) S2;
3 spin_lock(...);
4 ...
5 spin_lock(...);
```

Four potential paths can reach the double-lock, depending on the valuations of e_1 and e_2. But the case in which both e_1 and e_2 evaluate to `false` involves less steps, since neither S_1 nor S_2 are executed. When `if` statements do not have an *else* alternative, which is common, choosing the *false* valuation can lead to simpler bug traces.

6.4.2 Detection of spurious aliasing

Alias analysis is imperfect and, under some circumstances, the same regions may be assigned to two different objects, making them appear to be the same. EBA relies on aliasing information to find bugs across function calls and, as a result, spurious aliasing relations can lead to false bug candidates. One way of dealing with these false matches would be to perform a more careful analysis of the bug trace. For now, EBA relies solely on heuristics. The two main heuristics used to recognize false double-lock bugs caused by imprecise aliasing information are detailed below.

Different struct members. Lock objects are often part of a struct, and they are used to protect accesses to certain members of the struct. If EBA finds a double-lock where the two lock acquisitions are performed on different struct members, as in the example below, it considers the bug candidate spurious, and removes it. This form of aliasing is unlikely to happen in practice.

```
1 mutex_lock(x->mutex);
2 ...
3 mutex_lock(y->ca_mutex);
```

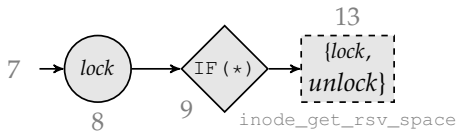
Updates between accesses. One of the major limitations of the shape inferrer is that it *flattens* data structures such as arrays and linked lists. All the elements in the data structure are given the same memory regions, and hence are indistinguishable from the perspective of the alias analysis. This leads to false positives when traversing data structures:

```
1 for (int i=0;i<N;i++) {
2         spin_lock(&a[i]);
3 }
```

In this example, the model-checker will warn that two iterations of the `for` loop could lead to a double-lock. EBA recognizes that this may not be a true bug because in between the two accesses to `a[i]`, the variable `i` is updated by `i++`. Most likely the two accesses are fetching different objects, and thus the bug candidate is removed. So, as a general rule, for EBA to report a double-lock bug, the l-value representing the lock object shall not have been affected by any updates in between the two acquisitions.

6.4.3 Abstraction refinement

Function summaries tell EBA whether a function call has no interesting effect. For instance, in Fig. 1.2, a call to a function that does not acquire nor release `inode->lock` would have effects $\emptyset$, and would be ignored. But function summaries are not enough for claiming that a bug exists, because they do not tell in which order the effects happen, nor under which conditions. For instance, in Fig. 1.3, the first check for bugs in `add_dquot_ref` (where `inode_get_rsv_space` is treated as a black-box) will result in the following bug candidate:



Yet, in this match, the second lock acquisition happens at node 13, which is a call to `inode_get_rsv_space`. As reflected in its signature, function `inode_get_rsv_space` both acquires and releases the lock, but the order of these operations is unknown when model-checking `add_dquot_ref`. In such case, the effect-abstraction of `add_dquot_ref` can be *refined* by inlining the call to `inode_get_rsv_space`. (Note that function calls that manipulate no locks, or manipulate locks different than the one being tracked, will never be inlined.)

Then, the bug-filter instructs the model-checker to resume the search on the refined *F*-CFG and a new match, this time conclusive, is found

(cf. Fig. 6.2). This inlining strategy is a simple form of *Counter Example Guided Abstraction Refinement* (CEGAR) [CGJ+00]. It allows EBA to support precise inter-procedural bug finding with a very simple effect language—which otherwise would have to capture ordering.

Chapter 7

Evaluation

This chapter evaluates the bug finding technique proposed in Chapter 6, through the `OCaml` implementation, `EBA`. The objective is to assess both the *effectiveness* and the *scalability* of this bug-finding technique.

7.1 Method

Effectiveness and scalability are fairly subjective terms without a baseline. For instance, a static analyzer that proves the absence of runtime errors in a 100 KLOC codebase in about ten hours can be considered scalable. The same would not be tolerable for a linter or bug finder. The same applies to effectiveness: static analyzers shall find every single bug, whereas bug finders are good finding *most common* bugs.

In short, we would like to answer the following research questions:

RQ1 How does `EBA` compare to similar bug finding tools?

RQ2 How precise is `EBA` on a large and complex codebase?

This evaluation emphasizes the comparison with well-established bug finders, which are used as baseline. In order to evaluate the scalability of bug finding tools it is necessary to test them on complex and large pieces of software. `Linux` is a good codebase for this: it is large—more than 10 MLOC, and uses all sorts of `C` constructs and idioms, making

it very complex to analyze. Besides, Linux is a popular project that has traditionally attracted the attention of the static analysis community, and there exist several Linux-tailored bug finders we can compare with.

Thus, the evaluation consists of two experiments, where the performance of EBA is measured in terms of analysis time and bugs found. EBA is compared against similar bug-finding tools: (1) on a benchmark of historical Linux bugs (RQ1, cf. Sect. 7.2); and (2) on the set of device drivers shipped with Linux-4.7 (RQ2, cf. Sect. 7.3). The first experiment tries to build an argument for the *effectiveness* of EBA: does it find more bugs than other similar tools? The second experiment focus on *scalability*: is EBA useful for analyzing a large amount of complex C code?

For simplicity, the experiments only target one type of bug: *double locks*. The technique is general and should find other types of bugs (cf. Sect. 3.8). But, at this stage, adding more bug types has a significant implementation cost in terms of modeling features (e.g., to inject domain-knowledge into EBA). Also, finding out the best heuristics to remove false positives. Finding double-locking bugs requires little extra infrastructure, and these bugs are a good representative of resource mismanipulation.

In fact, some studies suggest that locking bugs may represent 30% of bugs in OS code [CYC+01]. Double locks are introduced regularly into device drivers, due to the existence of multiple entry points to the drivers code, and have bad consequences for the user (i.e., the device hangs). Double-lock checkers are also part of many research tools that have used the Linux kernel for evaluation [FTA02, XA05, PTS+11], making it easier to find tools to compare with.

7.1.1 Subjects

This evaluation will compare EBA against Smatch and Coccinelle (cf. Sect. 2.3.2), two well-known tools among Linux kernel developers. Smatch is a bug finder by Dan Carpenter, it is an intraprocedural flow-analyzer similar to early versions of xgcc/metal [ECCH00], that is now funded by Oracle. Coccinelle is a program matching and transformation tool by Julia Lawall and others, currently developed at LIP6/INRIA. Coccinelle has a powerful matching engine based on CTL model checking, that can also be (and often is) used for bug finding purposes [CWY+13].

I selected these two baseline tools for two reasons. First, they are able to run out-of-the-box on the source code of Linux, without major

adaptation or further research. Second, there exist double-lock checkers tailored to the Linux kernel available for both of them. Smatch includes a Linux-specific double-lock checker built-in, and a double-lock checker for Coccinelle is shipped with the Linux distribution.[1]

Neither CppCheck, nor Clang Static Analyzer, nor INFER ship with a double-lock checker, so they could not be used for an independent comparison. Writing a *good* checker for any static analysis tool requires certain knowledge about the tool internals, and doing it myself would inevitably introduce bias. Sparse and CQual can detect double locks, but both require modifications to the analyzed source code, in the form of annotations. Finally, I had to exclude Saturn, which I could not build against a recent version of OCaml.

7.1.2 Reproducibility

Evaluation artifacts and detailed instructions are available online.[2] All experiments have been conducted on a virtualized machine with a physical 8-core (16-thread) Intel Xeon E5-2660 v3 CPU, running at 2.6 GHz and with 16 GB of RAM.

7.2 Performance on a benchmark of historical Linux bugs

7.2.1 Setup

I evaluate the EBA tool on a benchmark of 26 double-lock bugs extracted from historical bug fixes in the Linux kernel. In order to simplify the execution of the experiment, the benchmark is made of individual C files, each one containing (at least) one *known* bug. The experiment consists in running the three tools on these files, and measuring the analysis time and the recall (i.e., whether the bug is found).

In establishing this benchmark, I first obtained a set of 77 candidates by selecting all commits containing the phrase *"double lock"* in its message.[3] I filtered out 30 cases that were false positives (i.e., commits not fixing a double-lock bug), and 18 cases that were bugs spanning multiple files. To avoid bias, I removed two commits (3c13ab1 and 1d23d16) that were fixes to bugs found by EBA earlier during the development of the

[1] Located at `scripts/coccinelle/locks/double_lock.cocci`.
[2] `https://github.com/iagoabal/2017-vmcai`
[3] Extracted from the Linux kernel's Git repository as of August 3, 2016.

prototype. However, I kept any bug-fix derived from the two baseline tools, which introduces bias in favor of Coccinelle and Smatch.

For the 27 remaining commits, I obtained a preprocessed version of the file where each of these bugs were located, under the 64-bit x86 *allyes* configuration. This step excluded one file (commit 553f809) that failed to preprocess under this configuration. For Coccinelle, I retained the original source file, since it is designed to run on unprocessed C files. I then verified that the alleged bug was indeed present in the preprocessed file. Thus, I arrived at a benchmark of 26 double-lock bugs derived from historic Linux bugs. This benchmark is available online. [4]

7.2.2 Results

Table 7.1 shows the results of running EBA, Smatch, and Coccinelle on this benchmark. Each bug is identified by the commit that fixes it, and bugs are grouped by *depth*. The depth of a bug corresponds to the number of nested function calls involved from the first to the second acquisition of the lock. For instance, the bug of Fig. 1.2 involves one nested function call and therefore has depth one. The three rightmost columns of Tbl. 7.1 give the total analysis time (in seconds) for each of the three tools. For instance, for the first bug in the table, 00dfff7, EBA takes five seconds and correctly reports the bug. Smatch and Coccinelle take 1.5 and 0.1 seconds respectively, yet are unable to find the bug (hence the gray strikeout font).

Effectiveness. The first thing to point out is that EBA finds 22 out of the 26 bugs. In comparison, Smatch and Coccinelle find 14 and 12 bugs respectively. Note that this benchmark is biased in favor of the baseline tools, since many of the bugs were in fact reported by these two tools. Further, all bugs can be found without including headers, which is an advantage for Coccinelle—it has optional support for including headers but then it makes it significantly slower.

> *On this benchmark,* EBA *is significantly more effective (46%) at finding double-lock bugs than the two baseline tools combined.*

[4] https://github.com/IagoAbal/2017-vmcai/tree/master/5.1/

In particular, EBA finds six out of the nine interprocedural bugs (depth one or more), whereas Smatch and Coccinelle do not manage to find any at all. EBA has been designed with the goal of finding interprocedural bugs efficiently. Smatch does offer limited inter-procedural support for some checkers, but—as also confirmed by the author—not for the double-lock checker. For the remaining 17 intraprocedural bugs (depth zero), EBA finds all but one (16 out of 17), while Smatch and Coccinelle find 14 and 11,

Table 7.1: Comparison of EBA, Smatch, and Coccinelle on 26 historical double-lock bugs in Linux. Times in gray strikeout font indicate that the bug was *not* found by the tool.

Bug		Time (seconds)		
hash ID	depth	EBA	Smatch	Coccinelle
00dfff7	2	5.0	~~1.5~~	~~0.1~~
5c51543	2	2.3	~~1.5~~	~~0.3~~
b383141	2	~~6.1~~	~~2.9~~	~~0.3~~
1c81557	1	5.0	~~1.9~~	~~0.6~~
328be39	1	8.9	~~1.7~~	~~0.2~~
5a276fa	1	~~0.9~~	~~1.2~~	~~0.2~~
80edb72	1	~~6.3~~	~~2.1~~	~~0.7~~
872c782	1	1.7	~~2.8~~	~~1.9~~
d7e9711	1	21	~~1.3~~	~~2.7~~
023160b	0	1.0	2.6	~~0.1~~
09dc3cf	0	1.2	~~1.4~~	~~0.1~~
0adb237	0	1.1	1.5	0.2
0e6f989	0	0.4	1.0	0.3
1173ff0	0	0.6	1.3	0.1
149a051	0	~~0.7~~	~~0.6~~	~~0.3~~
16da4b1	0	0.4	0.8	0.1
344e3c7	0	0.7	1.3	~~0.1~~
2904207	0	5.8	2.0	~~2.8~~
59a1264	0	0.2	0.6	0.1
5ad8b7d	0	0.6	3.4	0.1
8860168	0	0.7	1.0	0.1
a7eef88	0	0.6	1.2	0.2
b838396	0	3.3	2.8	1.1
ca9fe15	0	0.4	~~0.7~~	1.8
e1db4ce	0	0.4	1.1	0.2
e50fb58	0	0.5	0.9	0.1

respectively. Remarkably, any bug found by either Smatch or Coccinelle, is also intercepted by EBA.

False negatives. EBA misses five bugs mainly due to limitations of the pointer analysis (cf. Chapter 4). For instance, in bug 80edb72, the lock object is contained in a `mv88e6xxx_priv_state` struct, and the latter is obtained from a `struct dsa_switch` pointer, after a non-trivial type cast involving pointer arithmetic. This case falls outside what the inference system is designed to handle (cf. Sect. 4.6) and the same lock, obtained at two different program locations, is seen as two different locks. For Smatch, false negatives seem to be due to path-insensitivity and lack of inter-procedural support. Coccinelle lacks inter-procedural support and, in addition, its double-lock checker does not recognize some common locking functions—otherwise it may have found more bugs than Smatch.

For instance, Smatch does not recognize bug 09dc3cf, presumably because the point at which the second lock occurs can be reached in many different ways, with both the lock held and unheld. It is common that, in such a case, an intraprocedural analysis will fall on the safe side and avoid reporting a (potentially false) bug. The same bug, 09dc3cf, is missed by Coccinelle because it does not "know" about the `raw_spin_lock_irqsave` function. Bug 149a051 is an interesting case, where function `as_merged_requests` takes two `struct request` pointers, and locks on both. If the two pointers point to the same object, a deadlock happens. The three bug finders make the assumption that the formal parameters of a function do not alias one another—as indeed mostly the case; and thus, all three tools missed that bug.

Analysis time. For the bugs that all three tools find, EBA is on average about 1.4 times faster than Smatch, yet Coccinelle is about five times faster than EBA. Note, however, that Smatch is checking for more bugs than double locks. (The CLI of Smatch does not allow to select specific bug checkers.) Also, EBA and Smatch analyze a total of 665 KLOC of *preprocessed* C code, whereas Coccinelle analyzes 27 KLOC of *unprocessed* C files. Preprocessed files typically contains large amounts of dead code from included headers. EBA does not analyze dead code by default, and presumably neither does Smatch.

Table 7.1 shows that variance of execution times is higher for EBA, with six files taking more than five seconds to analyze, and one file taking 21 seconds. These files contain large functions that manipulate

multiple locks and, as of now, EBA will check one lock object at a time. EBA should speed up considerably with some optimization work: e.g., by checking the use of all locks in a single traversal. Each Coccinelle check requires an independent run of the tool, so if it had to scan files for multiple bug types, it would be considerably slower than both EBA and Smatch.

7.3 Performance of analyzing device drivers in Linux-4.7

7.3.1 Setup

In this experiment, EBA is used to analyze widely the set of device drivers shipped with Linux 4.7-rc1, in search of double spin lock bugs. Note that drivers constitute around 60% of the Linux source code. As in the previous experiment, the analysis is performed for the 64-bit x86 *allyes* configuration. EBA is invoked by Kbuild through a "fake" GCC wrapper, during a parallel build process with 16 jobs (i.e., `make -j16`). This wrapper extracts the preprocessor options from the command line arguments, uses (the real) GCC to preprocess the file, and forwards the output to EBA.

About nine thousand files in `drivers/` were analyzed, and every bug alarm was manually classified as either a true or a false positive. This process was repeated for Smatch and Coccinelle, in order to confront analysis times and number of false positives. All tools were given 30 seconds to analyze each file.

7.3.2 Results

Table 7.2 shows the results of analyzing Linux-4.7 `drivers/` with EBA, Smatch, and Coccinelle.

Bugs found. EBA raised nine alarms in nine different files. *Five* of these bugs have been reported, *confirmed* by the respective Linux maintainers,

Table 7.2: Analyzing the entire `drivers/` subsystem of the Linux kernel.

	EBA	Smatch	Coccinelle
Bugs found	**5**	*N/A*	*N/A*
False positives	**4**	8	6
Time (minutes)	23	16	**2**

and *three are now fixed* in Linux-4.10 (see commits 1d23d16, e50525b and bea6403).[5] These bugs affected five different subsystems—`tty`, `scsi`, `usb`, `iommu`, and `net`; and all of them have depth one or more, thus requiring an inter-procedural analyzer. Smatch and Coccinelle found no bugs, but that is somewhat expected because these tools are run extensively on Linux source code and, presumably, any bugs would have already been reported and fixed.

Analysis time. EBA analyzes all Linux drivers (under the aforementioned configuration) in *less than half an hour* (23 minutes) and is only slightly slower than Smatch which does the same in 16 minutes (1.4 times faster than EBA). Coccinelle is significantly faster and completes the analysis in only two minutes, as it scans much smaller unprocessed files. Despite EBA is not as mature as the two baseline tools, the data shows that EBA scales well and it can be classified as lightweight bug finder. The same comments made about the tools in Sect. 7.2.2 apply here as well.

> EBA *analyzes all* Linux *drivers in less than half an hour, and finds five confirmed bugs.*

False positives. EBA reported nine bugs—that is an alarm for 0.1% of the files analyzed, pointing to five real bugs, and just four false alarms. Thus, EBA can be used to write bug checkers that find real problems, while producing little noise. Both Smatch and Coccinelle report more false positives (eight and six, respectively). It is worth noting that, at least in eight of the 14 false positives reported by the baseline tools, there was an *unlock* operation being performed through a nested function call—that these tools were not aware of.

Two of the four false bugs reported by EBA, can be traced back to limitations in the pointer analysis. Note that these limitations can both lead to false positives, and false negatives—as for bug 80edb72 in Sect. 7.2.2. Another case was due to EBA making reasonable but unsound assumptions about the aliasing of formal parameters, similar to what caused a false negative for bug 149a051 in Sect. 7.2.2. However, in this case the

[5] Bug e50525b was independently found and fixed during beta testing, but that bugfix was unknown to me.

code was particularly confusing, even for a human inspecting the code, and this bug report still led to a cosmetic fix (see 3e70af8). In the remaining case, the reported error trace was not a feasible execution path.

Chapter 8

Conclusion

There is no definitive answer to the problem of program verification. Lightweight bug finders are becoming increasingly popular, they scale well and can find common programming errors, but the errors they find are superficial. Heavyweight static analyzers perform a deeper semantic-based analysis of code, and find more bugs and more complex, but are orders of magnitude slower. The objective of this work has been to push the current bug-finding technology a step further, in order to make these tools find more complex bugs, while retaining their good scalability properties.

Ultimately, this work has led to the development of a novel bug-finding technique. In order to ensure both the applicability and the impact of this technique, I have followed a problem oriented methodology. First of all, I studied tens of historical bugs in Linux (cf. Chap. 3), with the purpose of identifying the challenges of analyzing large and complex software systems (cf. Sect. 3.7). This analysis revealed that a significant amount of bugs are due to violations of simple API rules, such as "spin locks cannot be acquired twice before being released".

Even if simple, these bugs often span multiple functions, and are then missed by current bug finders. Inspired by the use of side-effect analysis in interprocedural program optimization [Ban78, Ban79, CK84, CK88], I devised a lightweight interprocedural bug finding technique based on analyzing side-effect abstractions (cf. Sect. 3.8). This thesis has served to elaborate and formalize such technique, and to show that it is capable of uncovering deep resource mis-manipulation bugs in large systems-level software.

The technique consists of two steps. First, a *shape-and-effect* inference system is used to build an abstraction of the program to analyze (cf. Chap. 4). In this abstraction, objects are described by memory *shapes*, and expressions and statements by their operational *effects*. Second, bugs are found by matching temporal bug-patterns against the control-flow graph of this program abstraction (cf. Chap. 6). I have implemented a proof-of-concept bug finder based on this technique, EBA, and confirmed that it is both scalable and effective at finding bugs.

I have compared the performance of EBA with respect to Coccinelle and Smatch, two popular bug-finders within the Linux community (cf. Chap. 7). On a benchmark of 26 historical Linux bugs, EBA was able to detect significantly more bugs, and more complex, than the other two baseline tools combined. EBA is able to analyze nine thousand files of device drivers from Linux-4.7 in less than half an hour, in which time it uncovered five previously unknown bugs.

Future Work

The way that the study of Linux bugs has been conducted was influenced by the objectives of the VARIETE project, in which this thesis is framed. The focus on variability has biased the selection of bugs. It would be beneficial to generalize this study, and gather a larger amount of bugs, without those having to be caused by feature interactions. The good acceptance of VBDb suggests that a wider effort of documenting bugs in open-source projects would be greatly appreciated, by the static analysis community as a whole. There is an evident lack of documented examples and benchmarks derived from real cases. Research in static analysis should be more problem oriented, in order to have a greater impact on the reliability of future software.

I am proud of every single bug that EBA has found. Nonetheless, this is just a proof-of-concept prototype. EBA was built to demonstrate that effect-based bug finding is effective,[1] but I would not like this argument to be reversed. The limitations of EBA are not necessarily limitations of the underlying technique. In particular, a major limitation concerns tracking aliasing through type casts, and accesses to arrays and dynamically linked data structures. The shape language, and consequently the inference algorithm, needs more work in this direction. While this is

[1] Pun intended!

a difficult problem, I believe that there should be approximations that would yield good results in practice.

Much of the criticism that EBA has received was related to having found only bugs involving double locks. I have no reason to believe that effect-based bug finding is in any way bound to double locks. In fact, my study of real bugs in Linux code suggests the opposite. (But you are right to be skeptical about this.) In any case, it is surely worth adding more bug checkers to EBA and, in particular, memory management and security checkers. EBA should be able to find the control-dominated subtypes of these two bug categories as well; e.g., the use of untrusted data without sanitizing it first. EBA may offer significant scalability improvements over current tools based on interprocedural data-flow analysis.

Impact

The study of 43 historical bugs in Linux, while of limited scope, has produced a diverse and well-documented collection of bugs, that exemplifies major research challenges in the area of static analysis. Performing this study has been fundamental in guiding the design of the bug-finding technique here presented. I do hope that others will find this study as valuable as I do, in shaping future static analysis techniques. Remarkably, VBDb has attracted much attention from the intersection of the configurable software and static analysis communities, and it is slowly becoming an accepted benchmark to test early prototypes of variability-aware tools[Din15, AKGN+15, AHBT+16, vR16, IMD+17].

EBA has demonstrated that efficient interprocedural bug finding is possible. Until now, most interprocedural bug finders have been based on the RHS algorithm [RHS95]. RHS performs interprocedural data-flow analysis, while using memoization to avoid repeating the analysis of a function under equivalent inputs. This prevents duplication of work, but it still requires exploring the entire control-flow graph of a program. For control-dominated bugs, this thesis has shown that an inexpensive side-effect abstraction of the program can be very effective at pruning the control-flow graph. EBA is able to find interprocedural bugs while rarely inlining a function call. EBA is effective and, so far, it has found *more than a dozen double-lock bugs* in Linux 4.7–4.10 releases, most of them already confirmed and many fixed.

Bibliography

[ABW14] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the Linux kernel: A qualitative analysis. ASE, 2014.

[ABW17] Iago Abal, Claus Brabrand, and Andrzej Wąsowski. *Effective Bug Finding in C Programs with Shape and Effect Abstractions*. VMCAI 2017. 2017.

[AHBT+16] Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. Mutation operators for preprocessor-based variability. VaMoS '16, 2016.

[AKGN+15] Jafar Al-Kofahi, Lisong Guo, Hung Viet Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Static detection of configuration-dependent bugs in configurable software. ICSE '15, 2015.

[AMS+17] Iago Abal, Jean Melo, Stefan Stanciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. A hundred variability bugs in systems software: A qualitative analysis. 2017.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.

[ASW+11] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. ASE 2011, Lawrence, USA, 2011. IEEE Computer Society.

[BA08] Suhabe Bugrara and Alex Aiken. Verifying the safety of user pointer dereferences. In *Proceedings of the 2008 IEEE*

Symposium on Security and Privacy, SP '08, pages 325–338, Washington, DC, USA, 2008. IEEE Computer Society.

[Ban78] John Phineas Banning. *A Method for Determining the Side Effects of Procedure Calls.* PhD thesis, Stanford, CA, USA, 1978. AAI7905815.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. POPL, 1979.

[BBH+09] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[BC05] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. FMCO, 2005.

[BDH+] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: Using temporal logic and model checking. POPL 2009.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, October 2007.

[BHS03] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A system for generating secure crash information. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 19–19, Berkeley, CA, USA, 2003. USENIX Association.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. PLDI '01, 2001.

[BNE16] Fraser Brown, Andres Nötzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 143–157, New York, NY, USA, 2016. ACM.

[BR01a] Thomas Ball and Sriram K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. PASTE, 2001.

[BR01b] Thomas Ball and Sriram K. Rajamani. The slam toolkit. CAV, 2001.

[BRT+13] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development*, 10, 2013.

[BTR+13] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. SPL$^{\text{LIFT}}$ - statically analyzing software product lines in minutes instead of years. In *PLDI 2013*, 2013.

[BTSR04] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. POPL, 2004.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Form. Methods Syst. Des.*, 35(3), December 2009.

[CD11] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *Proceedings of the Third International Conference on NASA*

Formal Methods, NFM'11, pages 459–465, Berlin, Heidelberg, 2011. Springer-Verlag.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 289–300, New York, NY, USA, 2009. ACM.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, December 2011.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. CAV '00, 2000.

[CHSL11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *ICSE*, 2011.

[CK84] Keith D. Cooper and Ken Kennedy. Efficient computation of flow insensitive interprocedural summary information. SIGPLAN 1984, 1984.

[CK88] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. PLDI, 1988.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs, 2004.

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Comput. Netw.*, 41(1), 2003.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, June 1996.

[CWY+13] Yu Chen, Fengguang Wu, Kuanlong Yu, Lei Zhang, Yuheng Chen, Yang Yang, and JunJie Mao. HPCC/EUC, 2013.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[Dar86] Ian F. Darwin. *Checking C Programs with Lint*. O'Reilly, 1986.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. PLDI, 2000.

[DBW15] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses (extended version). *CoRR*, abs/1503.04608, 2015.

[Din15] Nicolas Dintzner. Safe evolution patterns for software product lines. ICSE '15, 2015.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. POPL 1982, 1982.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, Berlin, Heidelberg, 2006. Springer-Verlag.

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. OSDI, 2000.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, January 2002.

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

[FFA00] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 175–198, London, UK, UK, 2000. Springer-Verlag.

[FJKA06] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 2006.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. PLDI, 2002.

[GG12] Paul Gazzillo and Robert Grimm. SuperC: Parsing all of C by taming the preprocessor. PLDI '12, 2012.

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? PASTE, 2001.

[IMD+17] Alexandru Florin Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Effective analysis of C programs by rewriting variability. *CoRR*, abs/1701.08114, 2017.

[JG88] Pierre Jouvelot and David K Gifford. The fx-87 interpreter. In *Computer Languages, 1988. Proceedings., International Conference on*, pages 65–72. IEEE, 1988.

[JG91] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. POPL '91, 1991.

[Joh78] S. C. Johnson. Lint, a c program checker. Comp. sci. tech. rep, Bell Labs, 1978.

[JT93] Pierre Jouvelot and Jean-Pierre Talpin. The type and effect discipline, 1993.

[JVWS07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of functional programming*, 17(01):1–82, 2007.

[JW04] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[KA08] Christian Kästner and Sven Apel. Type-checking software product lines - a formal approach. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, L'Aquila, Italy, 2008.

[KGR+11] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. OOPSLA '11, 2011.

[KKHL10] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: Toward type checking #ifdef variability in c. FOSD 2010, 2010.

[Kop97] Rainer Koppler. A systematic approach to fuzzy parsing. *Softw. Pract. Exper.*, 27(6):637–649, June 1997.

[KS08] Oleg Kiselyov and Chung-chieh Shan. Lightweight monadic regions. Haskell, 2008.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. CGO '04, 2004.

[Lei14] Daan Leijen. Koka: Programming with Row Polymorphic Effect Types. MSFP, 2014.

[LLH+10] Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. Finding error handling bugs in openssl using coccinelle. EDCC 2010, 2010.

[LMP09] Julia L. Lawall, Gilles Muller, and Nicolas Palix. Enforcing the use of api functions in linux code. In *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS '09, 2009.

[Lov10] R. Love. *Linux Kernel Development*. Developer's Library. Pearson Education, 2010.

[LSB+10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the linux kernel variability model. SPLC, 2010.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2), January 2015.

[Luc87] John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, 1987.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6), November 1994.

[MBW16] Jean Melo, Claus Brabrand, and Andrzej Wąsowski. How does the degree of variability affect bug finding? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 2016.

[McA96] David McAllester. Inferring recursive data types. Technical report, AT&T Labs, 1996.

[McM05] Kenneth McMillan. *Applications of Craig Interpolation to Model Checking*. ICATPN 2005. 2005.

[MDBW15] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105(C), July 2015.

[MFBW16] Jean Melo, Elvis Flesborg, Claus Brabrand, and Andrzej Wąsowski. A quantitative analysis of variability warnings in linux. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '16, 2016.

[MKR+16] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. ICSE '16, 2016.

[MRG13] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating preprocessor-based syntax errors. GPCE 2013, 2013.

[NDT+13] Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. Linux variability anomalies: what causes them and how do they get fixed? In Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors, *MSR*. IEEE / ACM, 2013.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. CC 2002, 2002.

[NN99] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances*, volume 1710 of *LNCS*. Springer, 1999.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[Pad09] Yoann Padioleau. Parsing C/C++ code without preprocessing. CC '09, 2009.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Pea13] David J. Pearce. A calculus for constraint-based flow typing. FTfJP '13, 2013.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-sensitive correlation analysis for race detection. PLDI, 2006.

[PLM06] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers. EuroSys, 2006.

[PS08] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. ASE 2008, 2008.

[PTS+11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. ASPLOS XVI, 2011.

[PWC91] Michael Platoff, Michael Wagner, and Joseph Camaratta. An integrated program representation and toolkit for the maintenance of C programs. ICSM '91, 1991.

[Rem93] D. Remy. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming*. MIT Press, 1993.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL, 1995.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[SB15] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, January 1998.

[SST13] Jiri Slaby, Jan Strejček, and Marek Trtík. ClabureDB: Classified Bug-Reports Database. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *LNCS*. Springer Berlin Heidelberg, 2013.

[Ste96a] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. CC, 1996.

[Ste96b] Bjarne Steensgaard. Points-to analysis in almost linear time. POPL, 1996.

[SY86] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 1986.

[SYRS16] Bhargava Shastry, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. Towards vulnerability discovery using staged program analysis. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 78–97, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 2014.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2, 7 1992.

[TLL12] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. ICSE 2012, 2012.

[TLSSP11] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, 2011.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1), 1990.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. POPL, 1994.

[vR16] Alexander von Rhein. *Analysis Strategies for Configurable Systems*. PhD thesis, University of Passau, 2016.

[XA05] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. POPL, 2005.

[YHR99] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. PLDI, 1999.

[YMZ+11] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. SOSP 2011, 2011.

[ZMM+15] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. K. Petrenko, and A. V. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. *Program. Comput. Softw.*, 41(1):49–64, January 2015.