STANSE: Bug-finding Framework for C Programs

Jan Obdržálek, Jiří Slabý and Marek Trtík

Masaryk University, Brno, Czech Republic
{obdrzalek,slaby,trtik}@fi.muni.cz

Abstract. Stanse is a free (available under the GPLv2 license) modular framework for finding bugs in C programs using static analysis. Its two main design goals are 1) ability to process large software projects like the Linux kernel and 2) extensibility with new bug-finding techniques with a minimal effort. Currently there are four bug-finding algorithms implemented within Stanse: AutomatonChecker checks properties described in an automata-based formalism, ThreadChecker detects deadlocks among multiple threads, LockChecker finds locking errors based on statistics, and ReachabilityChecker looks for unreachable code. Stanse has been tested on the Linux kernel, where it has found dozens of previously undiscovered bugs.

1 Introduction

During the last decade, bug-finding techniques based on static analysis have finally come of age. One of the papers to really stir interest was [2], showing that static analysis can efficiently find many interesting bugs in real-world code. This work eventually led to a successful commercial tool called Coverity [10]. Over the years, several other successful tools, like CodeSonar [9] or Klocwork [12], appeared. However, such fully-featured tools are neither free to obtain, nor is their code available (e.g. for developing new algorithms or tailoring the existing tools to specific tasks). The existing free tools are usually severely limited in what they can do (e.g. Uno [15], Sparse [14], Smatch [13]). One notable exception is FindBugs [5, 11], a successful tool working on Java code. Stanse is intended to fill this gap for the C language. It can be seen in two ways:

1. Stanse is a robust framework (written predominantly in Java) for implementing diverse static analysis algorithms. An implemented algorithm can be immediately evaluated on large real-world software projects written in C as the framework is capable to process such projects (for example, it can process the whole Linux kernel). An implementation of such an algorithm within the framework is called a *checker*.
2. As Stanse already contains four checkers, it can be also seen as a working static analysis tool.

The paper is structured as follows. Section 2 describes the functionality provided by the framework, while Section 3 is devoted to the four existing checkers.

In Section 4 we present some results of running Stanse on the Linux kernel. The last section summarises the basic strengths of Stanse and mentions some directions of future development.

2 Framework Functionality

The Stanse framework is modular and fully open. It is designed to allow static analysis of large software projects like Linux kernel. Furthermore it is aimed to reduce effort when implementing a new static analysis technique. Architecture of the framework is depicted in Fig. 1, and is more or less standard. The bug-finding algorithms are implemented as *checkers*, and will be described in more detail in Section 3. In this section we focus on the functionality of the framework itself, describing only the non-standard or for some other reason interesting features.

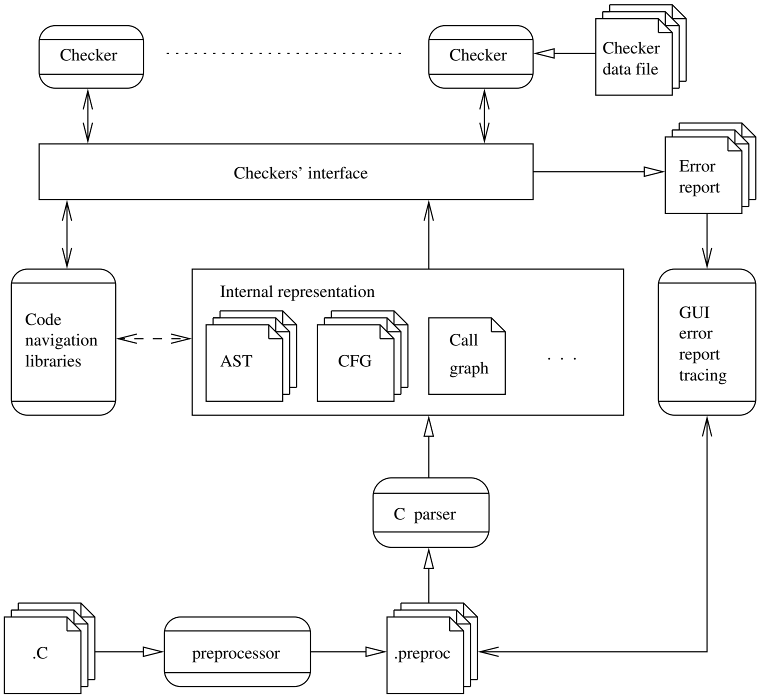


Fig. 1. Stanse framework architecture.

2.1 Configuration

The Stanse framework contains data structures capturing a configuration of an analysis to be performed. We always need to know what source files to analyse and by what checkers. That information we call *configuration*.

There are several ways how to tell STANSE which source files to analyse. Besides the standard possibilities (a given file, all files from a given directory, all files listed in a list), STANSE can also derive all the necessary information from a project Makefile. In this case, STANSE also remembers compiler flags for preprocessing purposes. This functionality is inspired by the SPARSE [14] tool.

The user must also specify the checkers which should be run on the configured source files. There can be many checkers running simultaneously in STANSE. However they cannot share any data and proceed independently. Checkers themselves can also be configured through their own configuration files. The configuration can be passed to the STANSE framework either via command line arguments, or using the graphical interface.

2.2 Parsing Source Files

The STANSE framework can process source files written in C, more specifically in the ISO/ANSI C99 standard together with most of the GNU C extensions. This allows STANSE to process software projects like the Linux kernel. We are currently working on support of other languages, in particular C++, a prototype implementation for which is included in the distribution. The parsing pipeline, including a preprocessing the source files by the standard GNU C preprocessor, is depicted at the bottom in Fig. 1. It is important to note that we do not parse the source files in a sequence one by one as they appear in a configuration. Since we use streaming (which we discuss later), the pipeline is applied as needed for each individual source file.

The parser used in STANSE is generated using the ANTLR tool from our own annotated C grammar. The reason for us to write our own parser was that at the time we started to develop STANSE we could not find free parser which, while being suitable for our purposes, would be able to parse most of the GNU C. (Linux Kernel makes heavy use of GNU C language extensions.) There is one notable exception: CLANG, which is slowly improving and will be able to handle the Linux kernel in the near future. However, in its current form it still contains bugs and cannot be used reliably. Our plan is to switch from our parser to CLANG once it becomes stable and feature complete.

Also, one may object that we could use the parser from the GNU compiler (GCC). Unfortunately this parser, and most importantly its internal structures, is not suitable for our purposes. For example the CFG is built on the top of RTL or tree representation (we encourage the reader to look into the GCC manual where these are described).

2.3 Program Internal Representation

Once the code is parsed, it is represented using STANSE's *internal structures*: a call graph among functions, a control flow graph (CFG) for each function, and an abstract syntax tree (AST) for the whole file. Subtrees in the AST are referenced from appropriate CFG nodes. We show these structures in the middle of Fig. 1. All these structures can be dumped in a textual or graphical form.

Since we aim at large software projects consisting of hundreds or thousands of modules (compilation units), it is often impossible in practice to store the corresponding internal structures in the memory all at the same time. The Stanse framework therefore applies automatic *streaming* of the internal structures. This is currently performed on a module basis. Instead of parsing all source modules in the beginning, a module is streamed in only when Stanse needs to access some internal structure belonging to the module. In other words, the internal structures are constructed on demand, in a lazy manner.

If the memory occupied by internal structures exceeds a given limit, some internal structures have to be freed before another module is streamed in. The structures to be freed are selected using the LRU (least recently used) approach: Stanse discards all internal structures of the module whose structures are not accessed for the longest time. If the discarded structure is accessed again later, the corresponding module is streamed back in. Both laziness of internal structures and streaming are completely invisible to checkers.

In the current implementation of streaming, each source file streamed out from the memory is completely discarded. Stanse does not back up already parsed internal structures into auxiliary files before discarding. As a consequence, when internal structures of the discarded file are needed again, Stanse starts the parsing pipeline of the file from scratch to recreate requested internal structures. Although loading of previously parsed internal structures from auxiliary files would speed up the streaming process, profiling of Stanse's performance on the Linux kernel has not shown streaming to be a performance bottleneck. However this could be easily changed if streaming performance becomes a problem in the future.

2.4 Pointer Analysis in Stanse

Since C programs tend to heavily use pointers, it almost always becomes a necessity to use some form of pointer analysis. There are many different known approaches to pointer analysis, differing in speed and accuracy. As each bug-finding/program analysis technique may have different requirements regarding pointer analysis, a framework like Stanse should ideally implement several different pointer analysis techniques and provide them to its checkers.

Nevertheless, the Stanse framework currently provides just two pointer analyses: *Steensgaard*'s [7] and *Shapiro-Horowitz*'s [6]. Both analyses are *may* analyses – they compute an over-approximation of an accurate solution. The *Steensgard*'s analysis is very fast and it is widely used in practice. On the other hand it is not very accurate. The *Shapiro-Horowitz*'s analysis allows parametrisation between *Steensgard*'s and *Andersen*'s analyses. One can therefore balance between speed of *Steensgard*'s analysis and accuracy of *Andersen*'s one.

2.5 Matching Language Constructs

Many static analyses change their internal state only on some subset of program expressions. For example, when finding race conditions in a parallel program, one may only focus on expressions involving synchronisation, while ignoring all others. The Stanse framework therefore provides a specification language for determining a set of program expressions.

The language defines a collection of *patterns*. Each pattern is supposed to identify a single specific kind of sub-trees in AST of analysed program. A pattern itself is therefore also a sub-tree of AST, where some of its vertices are "special". They allow to define a set of possible sub-trees at that vertex.

This is, however, not the only possible approach. For example, in the METAL [1] specification language, a `C` expression can be directly parametrised to define a set. The solution we implemented exploits the fact that checkers in Stanse work with AST intensively, and therefore identifying expressions in terms of AST is very practical.

2.6 Traversing Internal Representation

Although a checker may need to work with the internal structures in an arbitrary way, most checkers walk through CFGs using some standard strategy. To prevent unnecessary reimplementations, the most important and heavily used traversal methods are implemented directly inside the framework. With this functionality, one can implement a new checker (or its part) by specifying

- whether it should go through CFGs forwards or backwards, breadth-first or depth-first,
- whether the interprocedural walk-through should be performed or not (if not, the function calls are ignored), and
- a method (callback) to be called for each visited node in a CFG.

This makes implementation of new algorithms extremely simple.

For example, when a checker needs to implement a forward flow-sensitive analysis, it may ask the Stanse framework to traverse paths in CFGs in forward depth-first manner. This can be implemented by a single call to a function `traverseCFGToDepthForward`, which takes as an argument a CFG and a subclass of Stanse class `CFGPathVisitor`. In this class the checker defines the action which should be taken whenever a CFG node is visited (already in the requested order). The checker implements the action in a method `visit` of the subclass.

In addition, for those interprocedural analyses which do not construct summaries Stanse provides an automated traversal among different CFGs according to function calls (involving automated parameters passing and value returning). Again, this can be done using a single call to the Stanse framework.

The functionality described in this section is shown in Fig. 1 as "Code navigating libraries".

2.7 Support for Function Summaries

Interprocedural analyses typically build function summaries. Unfortunately, these summaries may differ from one analysis to another. Nevertheless, quite common part in building many summaries is passing formal and actual parameters to call sites and mapping return values to appropriate variables. Therefore, the Stanse framework provides classes simplifying the parameter passing and values returning for checkers. These classes also provide a conversion of a given expression in the caller into an equal expression in the called function. The conversion can also be required in the opposite direction, i.e. for returned values.

2.8 The Concept of Checkers

In the Stanse framework a checker is an implementation of some concrete static analysis technique. Each checker has an access to a shared internal structure of analysed source files. They are also provided with an access to the algorithms providing navigation in those structures. This is done by an interface between checkers and internal structure and libraries of the framework. The interface is depicted in Fig. 1 right bellow the checkers.

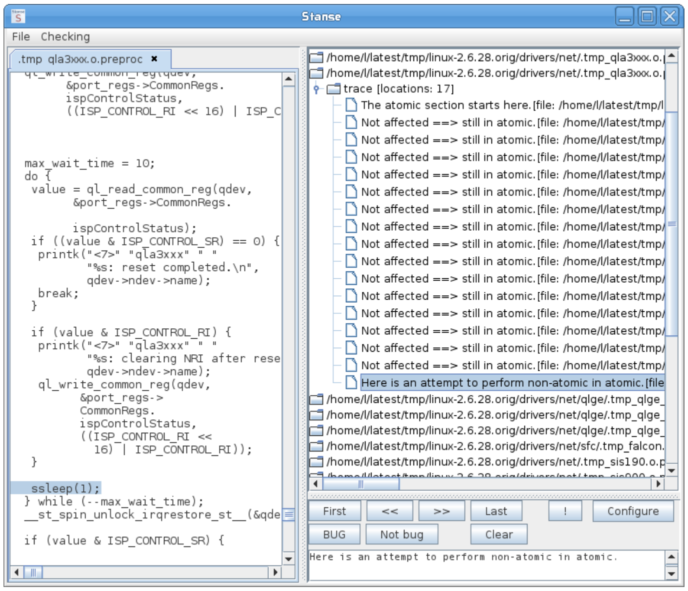
The checkers are integrated in the framework of Stanse using concrete factory design pattern. Therefore, to insert a new checker to the framework one needs to implement generic checker interface and register it to the checkers' factory of the framework. Then it gains a full access to the features of the framework accessible through the discussed interface.

It is very easy to integrate a new checker into Stanse. The process requires only three simple steps to be fully functional. The first step is to create a subclass of Stanse abstract class `Checker`, say `MyChecker`. The most important method to implement is `check`. There the analysis algorithm should be implemented.

The second step is to integrate the newly created class into the framework. This means implementation of `MyCheckerCreator`, a subclass of `CheckerCreator` abstract class. And the final step is to register the class `MyCheckerCreator`. It comprises adding a line `registerCheckerCreator(new MyCheckerCreator())` at the end of `CheckerFactory.java`.

2.9 Processing Errors

Once a checker finds an error, it reports the error back to the framework in the form of an annotated *error trace* - a path in the analysed code demonstrating this error. A datatype is provided in the Stanse framework to describe an error. In the framework there are then several possibilities how to present the error traces back to the user of Stanse: they can be printed to the console, displayed using a built-in error trace browser in the GUI (see Fig. 2), or saved to an external file in XML format. This XML file has a wide variety of possible applications. For example, we supply a tool transforming the XML file into an SQLite database. The database is supplemented with a web interface allowing to browse errors in the database via a web browser. Using the web browser or the built-in graphical error browser, one can mark errors as real bugs or false positives. Stanse also provides various statistics of errors like number of errors per checker, frequency of errors of the same kind, percentage of false positives (based on user feedback).



```
 ql_write_common_reg(qdev,
        &port_regs->CommonRegs.
        ispControlStatus,
        ((ISP_CONTROL_RI << 16) | ISP_C



 max_wait_time = 10;
 do {
  value = ql_read_common_reg(qdev,
         &port_regs->CommonRegs.

         ispControlStatus);
  if ((value & ISP_CONTROL_SR) == 0) {
   printk("<7>" "qla3xxx" " "
          "%s: reset completed.\n",
          qdev->ndev->name);
   break;
  }

  if (value & ISP_CONTROL_RI) {
   printk("<7>" "qla3xxx" " "
          "%s: clearing NRI after rese
          qdev->ndev->name);
   ql_write_common_reg(qdev,
         &port_regs->
         CommonRegs.
         ispControlStatus,
         ((ISP_CONTROL_RI <<
           16) | ISP_CONTROL_RI));
  }

  ssleep(1);
 } while (--max_wait_time);
 __st_spin_unlock_irqrestore_st__(&qde

 if (value & ISP_CONTROL_SR) {
```

```
/home/l/latest/tmp/linux-2.6.28.orig/drivers/net/.tmp_qla3xxx.o.
/home/l/latest/tmp/linux-2.6.28.orig/drivers/net/.tmp_qla3xxx.o.
  trace [locations: 17]
    The atomic section starts here. [file: /home/l/latest/tmp/l
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Not affected ==> still in atomic. [file: /home/l/latest/tmp/
    Here is an attempt to perform non-atomic in atomic. [file
/home/l/latest/tmp/linux-2.6.28.orig/drivers/net/qlge/.tmp_qlge_
/home/l/latest/tmp/linux-2.6.28.orig/drivers/net/qlge/.tmp_qlge_
/home/l/latest/tmp/linux-2.6.28.orig/drivers/net/qlge/.tmp_qlge_
/home/l/latest/tmp/linux-2.6.28.orig/drivers/net/sfc/.tmp_falcon.
/home/l/latest/tmp/linux-2.6.28.orig/drivers/net/.tmp_sis190.o.p
```

```
Here is an attempt to perform non-atomic in atomic.
```

Fig. 2. Error trace browser in the Stanse GUI.

Error reporting and tracing pipeline is depicted to the right of interface and internal representation of Fig. 1.

3 Checkers

In this section we briefly describe the four currently available checkers. All four checkers are provided with sample configuration so that they can be used instantly, however they can be configured differently when necessary.

AutomatonChecker is heavily influenced by [2]. It takes, as an input, a set of finite-state automata that describe the properties we want to check, patterns which match against the code to be checked, and finally transitions, i.e. pairing of patterns and automaton state changes. Properties like locking discipline, interrupt management, null pointer dereference, dangling pointers and many others can be described this way.

An example of the locking checker is presented in Fig. 3. The automaton starts in the unlocked state (U) and a transition is made when there is an outgoing edge from the current state with a pattern matching the action currently performed by the analysed program. E.g. if there is an unlock action while the automaton is in an unlocked state, an error is reported.

Compared to the implementation described in [2] and [4], our technique differs in several aspects. In particular, we do not use metacompilation, automata

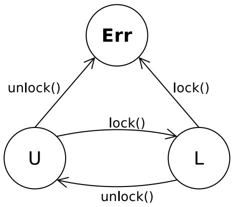


Fig. 3. Automaton for locks checking

are not input-language specific (a pattern matching is used instead), and the interprocedural analysis is done in the context of a single input file.

LockChecker accounts statistics about variable accesses. It also tracks which locks are locked while each of the variable is accessed. Again both variable accesses and locks are specified by patterns.

Then, combining the information about accesses and locks held, it counts a statistics in how many cases each variable is accessed while some lock is held. If the difference is proportional, an error is reported. So if, for instance, some variable is changed 99 times while some lock is held and in one case the lock is not, this is reported as a possible error. The boundary is currently set to 70, so that at least 70% of accesses must be under locks. The rest (30%) is then reported. This work is based upon [3].

ThreadChecker aims to check for possible deadlocks in concurrent programs. The technique is based on the notions of locksets of [8] and deadlock detection by looking for cycles in *resource allocation graphs (RAGs).* ThreadChecker first tries to identify the parts of the code which can run in parallel, as different threads. This is performed by searching of functions instantiating threads (such as `pthread_create`). Or, for the Linux kernel, we also specify manually which hooks may be run parallel.

Then the checker builds a set of *dependency graphs* for each such thread. A dependency graph statically represents possible locksets during one execution of a thread. Dependency graphs are then combined and transformed into RAGs. If there is a circular lock dependency, RAG contains a cycle. In such case an error is reported to the user.

ReachabilityChecker searches CFGs for unreachable nodes. These are then reported as warnings or errors, depending on importance (e.g. superfluous semicolons are less important than unused branch). The primary goal of ReachabilityChecker is to demonstrate the simplicity of a new checker implementation. With a help of the framework features described in Subsection 2.6, the code of the checker has less than 200 lines including the mentioned error/warning classification and many strings and comments.

Checker	**Automaton**	**Errors**			**Real/classified**
		Found	**Real**	**False pos.**	**error ratio**
AutomatonChecker	Pairing	266	65	143	31.3 %
	Pointers	86	48	37	56.5 %
	Deadlocks	35	16	18	47.1 %
LockChecker		13	6	7	46.2 %
ThreadChecker		20	9	11	45.0 %
ReachabilityChecker		31	31	0	100.0 %
Overall		**451**	**175**	**216**	**47.9 %**

Table 1. Stanse results on the Linux kernel version 2.6.28.

Even though it is a very simple checker it was still able to find serious bugs in the kernel. For example a superfluous semicolon can cause unexpected unconditional returns from functions like in the following code: `if (cond); return;`.

4 Results on the Linux Kernel

We have several reasons to choose the Linux kernel for testing Stanse: the kernel is a large and freely available codebase, it fully exercises most of the features of ISO/ANSI C99 and GNU C extensions, it is under constant development (there is a constant income of new bugs), and the absence of bugs is of a great concern.

We applied Stanse, together with the four checkers described in the previous section, to the Linux kernel version 2.6.28. The AutomatonChecker was configured with three automata describing the following types of errors:

- incorrect pairing of functions (imbalanced locking, reference counting errors)
- bugs in pointer manipulation (null dereference, dangling pointers, etc.)
- deadlocks caused by sleeping inside spinlocks or interrupt handlers

The running time of Stanse on a common desktop machine with two 2.5 GHz cores and 4 GiB of memory was under two hours. The memory usage of the Java process oscillated between 400 and 1000 MiB. The number of errors found by the checkers is presented in Table 1. Let us note that ReachabilityChecker actually found 751 errors, but 720 of them are of low importance (including 696 superfluous semicolons) and they are omitted from our statistics.

We have manually analysed all the found errors and classified them as real errors or false positives (with an exception of 60 errors found by AutomatonChecker where we are not able to decide in a short time whether it is a false positive or not). Note that the checkers do not produce any false negatives (assuming there is no bug in the checkers' implementation). The reason is that all the checkers implement may analyses, overapproximating the set of error behaviours. The numbers of real errors, false positives and the ratio of real errors to all classified errors can be also found in Table 1. The overall ratio of real errors to all classified errors is not high: 47.9%. However, Stanse in the current version does not have any thorough false positive filtering technique, which may be implemented in future.

More than 70 of the 169 real errors have been reported to kernel developers and fixed in the following kernel releases (the rest have been independently discovered and reported by someone else or the incorrect code disappeared from the kernel before we finished our evaluation of found errors). Some of the reported bugs remained undiscovered for more than seven years (for illustration, see our report at `http://lkml.org/lkml/2009/3/11/380`).

We have reported another 60 bugs found by Stanse in the subsequent versions of the kernel. This number is increasing every month.

4.1 Important Bugs found by Stanse

Although checkers currently implemented in Stanse are based on widely known techniques, running them on the Linux kernel helped to uncover several important bugs. In the text below we present two typical bugs discovered by Stanse, each using a different checker.

AutomatonChecker Many bugs found by the AutomatonChecker trigger only under specific conditions, however some of them may be visible to the user. Consider this code excerpt taken from the 2.6.27 kernel, drivers/pci/hotplug/pciehp_core.c file, set_lock_status function:

```
mutex_lock(&slot->ctrl->crit_sect);
/* has it been >1 sec since our last toggle? */
if ((get_seconds() - slot->last_emi_toggle) < 1)
  return -EINVAL;
```

Note that the call to mutex_lock function is followed by an **if** statement, which returns immediately in the true branch, omitting a call to mutex_unlock. In fact this deadlock could be easily triggered by a user. It is sufficient to write "1" to /sys/bus/pci/slots/.../lock file twice within a second.

ThreadChecker An example of non-trivial error which could not be found by the AutomatonChecker. The code described here is from the 2.6.28 kernel, file fs/ecryptfs/messaging.c.

There are three locks in the code, msg_ctx->mux, which is local per context, and two global locks – ecryptfs_daemon_hash_mux and ecryptfs_msg_ctx_lists_mux.

Let us denote lock dependencies as a binary relation where the first component depends on the second. I.e. lock(A) followed by lock(B) means dependency B on A, and we write $A \leftarrow B$.

```
int ecryptfs_process_response(...)
{
  ...
  mutex_lock(&msg_ctx->mux);
  mutex_lock(&ecryptfs_daemon_hash_mux);
  ...
  mutex_unlock(&ecryptfs_daemon_hash_mux);
  ...
```

```
 9  unlock:
10    mutex_unlock(&msg_ctx->mux);
11  out:
12    return rc;
13  }
```

Here the two locks on lines 4 and 5 give msg_ctx->mux ← ecryptfs_daemon_hash_mux.

```
14  static int ecryptfs_send_message_locked(...)
15  {
16    ...
17    mutex_lock(&ecryptfs_msg_ctx_lists_mux);
18    ...
19    mutex_unlock(&ecryptfs_msg_ctx_lists_mux);
20    ...
21  }
22
23  int ecryptfs_send_message(...)
24  {
25    int rc;
26
27    mutex_lock(&ecryptfs_daemon_hash_mux);
28    rc = ecryptfs_send_message_locked(...)
29    mutex_unlock(&ecryptfs_daemon_hash_mux);
30    return rc;
31  }
```

At line 28, function ecryptfs_send_message_locked is called from ecryptfs_send_message, hence the locks at lines 17 and 27 generate lock dependency of ecryptfs_daemon_hash_mux ← ecryptfs_msg_ctx_lists_mux.

```
32  int ecryptfs_wait_for_response(...)
33  {
34    ...
35    mutex_lock(&ecryptfs_msg_ctx_lists_mux);
36    mutex_lock(&msg_ctx->mux);
37    ...
38    mutex_unlock(&msg_ctx->mux);
39    mutex_unlock(&ecryptfs_msg_ctx_lists_mux);
40    return rc;
41  }
```

Finally, this function introduces ecryptfs_msg_ctx_lists_mux ← msg_ctx->mux.

Composing these results together the following circular dependency of these three locks was found:

- msg_ctx->mux ← ecryptfs_daemon_hash_mux
- ecryptfs_daemon_hash_mux ← ecryptfs_msg_ctx_lists_mux
- ecryptfs_msg_ctx_lists_mux ← msg_ctx->mux

This issue was later confirmed as a real bug leading to a deadlock[1].

5 Conclusions and Future Work

Stanse is a free Java-based framework design for simple and efficient implementation of bug-finding algorithms based on static analysis. The framework can process large-scale software projects written in ISO/ANSI C99, together

[1] http://lkml.org/lkml/2009/4/14/527

with most the GNU C extensions. STANSE does not currently use any new techniques – its novelty comes from the fact that (to our best knowledge) there is no other open-source framework with comparable applicability and efficiency. We note that more than 130 bugs found by STANSE have been reported to and confirmed by Linux kernel developers already. More information and the tool itself can be found at `http://stanse.fi.muni.cz/`.

Future Work We plan to improve the framework in several directions. Firstly we are currently working on C++ support. Furthermore we plan to provide STANSE in the form of an IDE plug-in, e.g. for Eclipse and NetBeans. A lot of work can be done in the area of automatic false alarm filtering and error importance classification. Independently of developing new features, we would like to speed up the framework as well. To this end we intend to replace the current parser written in Java by an optimised parser written in C, to replace the XML format of internal structures by a more succinct representation, to add a support for function summaries, etc.

Acknowledgements Jan Kučera is the author of the THREADCHECKER. We would like to thank Linux kernel developers, and Cyrill Gorcunov for STANSE alpha testing and useful suggestions. All authors are supported by the research centre Institute for Theoretical Computer Science (ITI), project No. 1M0545.

References

1. A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. *ACM SIGOPS Oper. Syst. Rev.*, 34(5):59–70, 2000.
2. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00*, pages 1–16, 2000.
3. D. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.
4. S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI'02*, pages 69–82. ACM, 2002.
5. D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA'04*, pages 132–136. ACM, 2004.
6. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL '97*, pages 1–14. ACM, 1997.
7. B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96*, pages 32–41. ACM, 1996.
8. J.W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *ESEC-FSE'07*, pages 205–214. ACM, 2007.
9. CODESONAR. `http://www.grammatech.com/products/codesonar/`.
10. COVERITY. `http://www.coverity.com/products/`.
11. FINDBUGS. `http://findbugs.sourceforge.net/`.
12. KLOCWORK. `http://www.klocwork.com/products/`.
13. SMATCH. `http://smatch.sourceforge.net/`.
14. SPARSE. `http://www.kernel.org/pub/software/devel/sparse/`.
15. UNO. `http://spinroot.com/uno/`.