Kayrebt: An Activity Diagram Extraction and Visualization Toolset Designed for the Linux Codebase

Laurent Georget Frédéric Tronel Valérie Viet Triem Tong
EPC CIDRE CENTRALESUPELEC/INRIA/CNRS/University of Rennes 1, Rennes, France
laurent.georget@irisa.fr, {frederic.tronel,valerie.viettriemtong}@centralesupelec.fr

***Abstract*—We present Extractor and Viewer, two tools from the Kayrebt toolset. The former is a plugin for the Gnu Compiler Collection (GCC) which builds pseudo-UML2 activity diagrams from C source code. It is specifically designed to handle the Linux kernel, a large and complex codebase. Use cases for this tool are numerous. The diagrams extracted from the C source code can be used to get a better insight of the control or data flow inside a program, or to evaluate the complexity of a function at a glance. Kayrebt::Viewer is a GUI designed for visualizing and navigating between the diagrams to explore source code.**

***Index Terms*—Activity Diagrams; Linux; Control Flow; Compilation; GCC.**

I. Introduction

As of version 4.0, published on April 13 2015, the Linux® codebase counted over 13,000,000 lines of code, most of which are device drivers. Not accounting for the drivers, the architecture-specific code and the common library functions, the overall total is roughly 3,000,000. (These statistics have been measured with cloc [1].) This represents a huge amount of code, which becomes quickly overwhelming for any single analyst wanting to get a complete understanding on a specific feature or to chase down a bug impacting several submodules of the kernel. Another problem is that although the Linux kernel obeys to some norms, such as the Single UNIX® Specification [2], which incorporates POSIX, those specifications are written in natural language and as such, can be incomplete or ambiguous, or even not properly implemented.

To tackle both those problems, the massiveness of the codebase and the lack of a proper semantics for some operations, we propose the Kayrebt toolset. This toolset is a collection of pieces of software designed to (1) extract knowledge from the C code under the form of activity diagrams, (2) equip those diagrams representing the code with a formal semantics (that we will not describe here), and (3) visualize the diagrams produced and navigate easily from any entry point in the kernel.The main target of our tools are people seeking to get a quick understanding of a feature without getting lost in the codebase. This is particularly useful when exploring architecture-specific details which makes the code intricated.

The Kayrebt toolset is composed of several independent yet collaborating tools. We often refer to the components of the toolset using the namespace-like notation Kayrebt::*Component* in this document. For the visualization part of the toolset, we can identify three major tools. The first one, Extractor, is a plugin for the Gnu Compiler Collection [3] (GCC). It is designed to extend the compilation process of GCC to extract activity diagrams out of the source code functions. The second one, Globsym, is used to build a database of functions exposed in a source code, with their declaration location. The third one, Viewer, is a Graphical User Interface that can be used to navigate between the activity diagrams generated for a project's codebase. We used GCC because Kayrebt is a toolset we built to analyse the Linux kernel. As of today, the official Linux kernel working branch (the one maintained by Linus Torvalds, responsible of the Linux project) relies heavily on some GCC peculiarities and does not compile entirely with other compilers such as LLVM. Of course, in our case, we focused on the Linux kernel because we built the Kayrebt visualizing tools for our specific needs but its conception is generic enough to be reusabe in other projects. In particular, any project that can be compiled with GCC can use Kayrebt, although the result is optimized for C code.

In this paper, we present two tools: Kayrebt::Extractor and Kayrebt::Viewer. The paper is organized as follows. In section II, we compare different tools used to extract and represent control flow graphs. Section III presents the design and development principles of our tools. Section IV presents an example of use of Kayrebt::Extractor whereas section V deals with the use of Kayrebt::Viewer. Finally, section VI concludes on our tools and give insight on future work.

II. Related Work

Several tools exist to dump the control flow graph in an activity diagram-like form out of source code but most of them are dedicated to static analysis rather than visualization. Although we did also built the Kayrebt toolset to perform static analysis on the Linux kernel (we will not enter into details in the present paper), we are also convinced that activity diagrams can be a useful representation of a source code if they can be automatically extracted from it and if they are navigable.

Interesting approaches include Moritz [4], an extension for the documentation tool Doxygen, and Crystal FLOW [5], among others. Contrarily to Flowgen, those tools generate flowcharts directly from the source code. Crystal FLOW and

other similar approaches let the user see the code and the flowchart side-by-side, which is really useful. Nevertheless, we have found those tools to be limited in the case of very large and complex code base such as Linux, which makes a heavy use of macros. They present exactly the code written by the developer, but in the case of Linux, because the code is highly configurable at compile-time, this representation is not always useful. For example, from all the code composing the entire Linux codebase, only a small subset usually ends up in the compiled kernel because the code corresponding to optional features can be compiled out. To sum up, they are an exact representation of the code written by the programmer instead of a view of what is actually compiled by the compiler.

Another approach is Flowgen [6], a tool able to extract activity diagrams from annotated C++ code. It is designed to make high-level diagrams using specially-crafted comments in the source code. Its main goal are documentation and easier collaboration between specialists of the problem dealt with by the program and skilled programmers, specialized in optimizing. Although this approach is very interesting, our work is quite different. With Kayrebt, we do not want to rely on human comments but on the *compiler*'s understanding of the code.

III. Development

Kayrebt::Extractor is a plugin designed and built for GCC 4.8 and Kayrebt::Viewer is a graphical Qt 4.8 application. They are both developed in C++11.

A. GCC Internals

GCC is organized in compilation steps called *passes*. Each step takes the output of the preceding step as input and process it to produce a lower-level, more optimized, further from the original source code and closer to the machine level, representation of the code, until the final machine code. A large part of the compilation process is independent both from the input language and target architecture and consists in optimizations and rewriting. GCC works on a per function basis. Functions are first optimized locally and interprocedural optimizations occur later in the compilation process. There is no optimizations across compilation units at all by default, although Link-Time Optimizations (LTO) can be optionally enabled.

GCC has a fairly recent and rapidly-changing support for plugins. The simplest class of plugins implements one or several passes, which have to be inserted in the compilation process, without changing the behaviour of other passes. Plugins are typically used to implement new optimizations or, like Kayrebt::Extractor, to instrument GCC and make dumps of internal representations. Depending on where the new pass is inserted, the internal data structures representing the code under compilation differ. We implemented one main pass to compute the activity diagram for the function under compilation. We have several constraint on the position of the pass. First, we want to represent only the code which is actually executed (no dead branch, etc.). We also want the code to remain human-readable (we want to avoid optimizations that change the control flow too deeply such as loop unrolling for instance). Finally, we want the flow graph to appear clearly in the representation so that the activity diagram is a simple translation of the Control Flow Graph (CFG) created by GCC.

B. Kayrebt::Extractor's Pass Placement

The pass building the CFG in GCC is called `cfg`. Hooking the plugin after this pass is interesting because it makes the construction of the activity diagram easier. The CFG representation is maintained until the very last passes. However, the code representation becomes quickly too far from the originally human-written code to be considered a good visualization. A few passes after the construction of the CFG, the code is transformed into Static Single Assignment (SSA) form as described in [7], [8]. In SSA form, every single variable is assigned once and once only. This allows several optimization through alias analysis, etc. In our case, this representation is a problem because transforming the code so that it conforms to the SSA property means modifying it too much to be a usable visualization of the original code. So, we chose to hook our plugin just before the change to SSA form. This tradeoff between the proximity to the compiler's vision of the code and the understandability for the user is the key idea of Kayrebt's design.

Optimizations made before this point include some dead code elimination, which is useful for us. Furthermore, there are no complex operations involving long arithmetic or boolean expressions at this point, all the code is in a three addresses form. One problem is that even before the compilation *per se* starts, the code is preprocessed: macros are replaced, files are included in the compilation unit, etc. On one hand, this is a good thing because macros can be used to extend the syntax of the language in an uncontrolled manner and unrolling them is necessary to represent the actual code, i.e. the code the compiler is going to process. On the other hand, macros are heavily used in C projects to manipulate constant values. Replacing their name by their value makes the diagrams less readable, especially if the value is meaningless (error codes fall in this category for instance).

C. Plugin Architecture

Extractor relies on a external library, called libactdiags (also part of the Kayrebt toolset), which is usually packaged along with Kayrebt::Extractor. This library contains the representation of the activity diagrams as well as methods to build them incrementally and output them in a specified format. For now, the only format available is Graphviz [9]. The plugin is divided in three modules: the entry point, the administrative module, and the plugin core. The plugin entry point is the interface with GCC. Its role is to install the new pass in the GCC compilation process. The administrative module is composed of a Configurator class handling the configuration of the plugin (which can be specified by the command line or via a configuration file) as well as classes that improve the activity

diagrams generated. Finally, the core is composed of classes mapping the CFG built by GCC for its normal compilation process into an activity diagram.

IV. Activity Diagrams Extraction with Kayrebt::Extractor

Kayrebt::Extractor provides the user with a view of the software under development or analysis. This view can be customized through configuration. The configuration is a simple YAML file. The listing 1 is an example of valid configuration. The configuration is split up in various sections: one section called "general" and one section for each compilation unit. Kayrebt::Extractor can function in two modes. In the greedy mode, it will generate an activity diagram for every function it meets. In the non-greedy-mode, the functions to graph have to be specified for each compilation unit. It is possible to configure the location of the configuration file by passing an argument to GCC for the plugin. Extracting one or several activity diagrams from the code under analysis is done by compiling it using GCC with the Kayrebt::Extractor plugin enabled.

Listing 1. Example of Configuration File

```
general:
  greedy: 0
  url:
    dbfile: 'my_db.sqlite'
    dbname: 'symbols'
  categories:
    1: 'bgcolor=blue'
    2: 'textcolor=red'

source_file1.c:
  functions: ['function1','function2']
  match:
    '(k|m)alloc.*': 1

source_file2.c:
  functions: ['one_more_function']
  start_match:
    '.*spin_lock.*': 2
  end_match: ['spin_unlock']
```

For the sake of the example, say that we are working on the Linux kernel and want to examine the system call `shutdown` which is used to shut a socket down and free some resources. We want to see more specifically the flow control relative to error handling. Assuming we start the compilation from the source tree's root, the file to compile (the file in which the system call `shutdown` is defined) is `net/socket.c`. The system call function is `SYSC_shutdown`. The configuration file is shown on listing 2.

Listing 2. Configuration File for the Extraction of Syscall `shutdown`

```
general:
  categories:
    1: 'style=filled,fillcolor=yellow'

net/socket.c:
  functions: ['SYSC_shutdown']
  match:
    '.*err.*': 1
```

The next step is to compile `net/socket.c` with the plugin enabled. As we are working in the Linux tree, we must respect the compilation chain and use the Makefile. The following command makes the compilation:

```
make CFLAGS_KERNEL='-fplugin=kayrebt_extractor
-fplugin-arg-kayrebt_extractor-config_file=
config' net/socket.o
```

This creates a file `net/socket.c.dump` containing the definition of the diagram for `SYSC_shutdown`. It is then possible to extract the diagram from this file and convert it into an image, for example using GraphViz's `dot` tool. Fig. 1 shows the resulting activity diagram. As one can see, it is possible to use the "categories" mechanism to embed any kind of attributes in the graph. Here, we chose to put in category 1 the nodes matching the regular expression '.*err.*', to highlight the use of this parameter, and to give background color yellow to category 1.

The collection of diagrams is useful for some tasks, like comparison between versions. In an activity diagram, the control flow is clearly visible, so even a predicate added in a existing if-else structure would be clearly visible. Conversely, the diagram also highlights the *absence* of something. For example, it is easy to distinguish a predicate like `ptr && ptr->inside` from `ptr->inside`, where the latter version could be a typical case of dereferencing a pointer without verifying it, because the former will result in a additional branching in the activity diagram.

V. Navigating between the Activity Diagrams with Kayrebt::Viewer

A. Presentation of Kayrebt::Viewer

For a better visualization experience, we built a Graphical User Interface (GUI) called Kayrebt::Viewer shown on Fig. 2). A thorough exploration of Kayrebt::Viewer is available as a screencast.[1]. In this section, we summarize the design and the main features of the program.

The main advantage of Kayrebt::Viewer over a simple collection of diagrams as images is that diagrams may be linked together. In a diagram, if a node representing a function call contains the attribute "URL" with a non-empty string, this string is understood as the path where the called function diagram is stored. This visualizer is project-centric. From a codebase, the user can generate all the diagrams she wants using Kayrebt::Extractor. A database of functions in the project is also required. It is a simple table with three columns required: `symbol` which gives the name of the function, `dir` the path to the directory where the compilation unit the function is declared in resides, and `file` the name of the compilation unit. Kayrebt::Viewer is parameterized through three paths: (1) the source directory of the project, (2) the database file, and (3) the directory where all the generated diagrams live. Those settings are persistent across executions of Kayrebt::Viewer.

[1] https://www.lgeorget.eu/code-panel/2015/05/31/kayrebt_viewer/; also available at https://youtu.be/Z94jgINyU3E

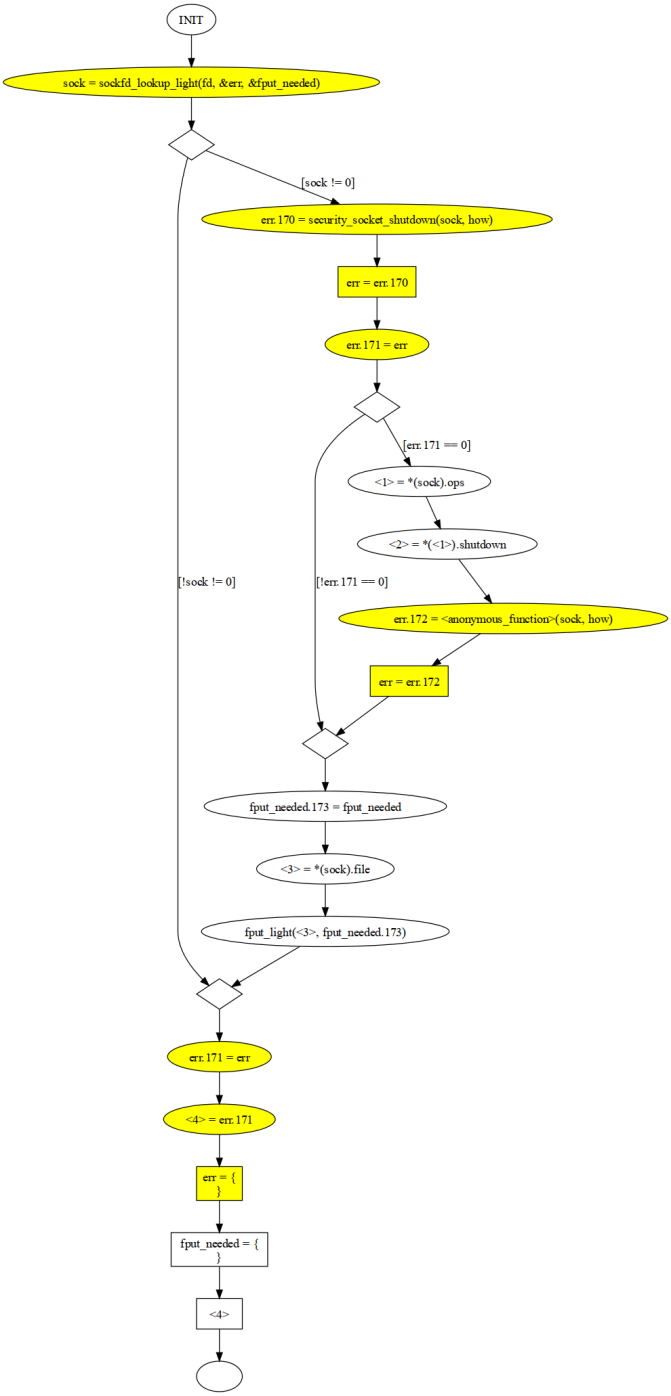

Fig. 1. Activity diagram of `shutdown` system call

The main window of the program is divided into three panes as shown in Fig. 2. The central and main pane is where the diagrams are displayed. Kayrebt::Viewer is a multi-documents interface, that is, an unlimited number of diagrams may be opened at the same time. The left pane is a tab widget. It contains a view of the symbol database presented earlier and an history of the diagrams visited, organized as a tree. Finally, the right pane shows a tree view of the source directory, as

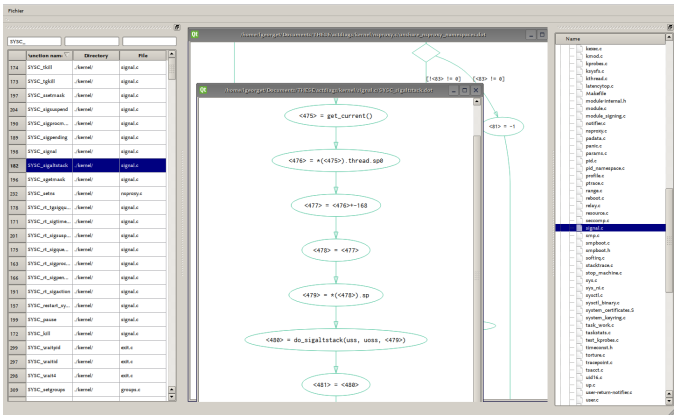

Fig. 2. Screenshot of the Kayrebt::Viewer's main window

configured. The database panel features multi-filtering and sorting ; this is useful to quickly look for an entry point in the kernel. We made the choice to include only exported functions in this database in order to limit the quantity of information displayed to the user.

B. Example of Use

One of the use case that lead us to develop the Kayrebt toolset was the necessity to track down the use of a given API in the kernel code, from a given entry point. For example, let us assume that we are exploring the system call `kill`, which is used by processes to send signals. We want to know when security checks are done and where the Linux Security Framework (LSM) has a callback point in the execution of `kill`. This example is presented in the screencast too but we summarize it here.

Typing `kill` in the filter in the left pane yields the interesting entry of the database: a symbol named `SYSC_kill`, in directory `kernel/`, in file `signal.c`. Double-clicking on it make the diagram appear in the central pane. It also has an action on the right pane: it selects and scrolls down to the file `signal.c` in the source tree, which can be useful in case the user might want to locate the file quickly to open it in an IDE at the same time. The function seems to be pretty straightforward. A structure called `info` is filled in with values and at the end a function `kill_something_info` must do the real work. In Kayrebt::Viewer, all function calls contain hyperlinks to the diagram of the function called. Ctrl+clicking on the node representing the function call opens the corresponding diagram. Note that the history is also updated at this instant to reflect the exploration path we have followed so far.

The new function that opened looks more complicated. Hovering the branching nodes with the mouse clearly highlights the reachable nodes. The nodes for which there exists a path from the hovered node are temporarily drawn with thick, red line until we move the cursor out of any node. This gives a quick overview of the control flow. Here, we see several successive branchings and a loop structure. The top level branching shows that a simple condition leads to two unrelated behaviours:

whether the PID of the process to kill is positive or not. With the help of the manual page of `kill`, it appears that if the PID is positive, then the signal is sent only to one process, otherwise, it is sent to a group of process. If we want to focus on the first case, it is possible to remove branches from the diagrams. This is achieved by double-clicking on a node or an edge: the whole branch (i.e. the nodes and edges accessible only from the node or edge we double-clicked) is removed. Right-clicking on the diagram makes a contextual menu appear, with a button to reset the diagram.

Removing the right branch leaves us with very few nodes actually. We see a section enclosed with `rcu_read_lock` and `rcu_read_unlock`, which are two primitives to synchronize the access on a kernel data structures. Between these two nodes, a function looks promising: `kill_pid_info`. This function is still in file `signal.c`, this can be verified in the history view. It is simple, there is a loop around an `rcu_read_lock` / `rcu_read_unlock` section, and inside there are two function calls `pid_task` and `group_send_sig_info`. We can deduce that here, the pointer to the process with the parameter PID is fetched, and then the signal is sent using it. In the case the task fetched is invalid (because of a concurrent access from another thread, for instance), an error will be returned and the control flow will go back to the beginning of the loop. This can be inferred with additional knowledge : the functioning of the *Read Copy Update* (RCU) mechanism.

Function `group_send_sig_info` is rather simple. We find one interesting function at the beginning: `check_kill_permission` and another one later on, on a branch which can be taken only if the check returns 0 and the signal to send is not 0: `do_send_sig_info`. Presumably, the former does all the sanity and security checks on the arguments and the latter actually send the signal. This is correct according to the `kill` manual which says that a signal 0 can be used to check if the permissions of the current process are sufficient to send a signal to the target without actually sending a real signal. The two functions can be explored. It is always possible to get back to an ancestor function using the history to jump from one diagram to another. A diagram already open, is not drawn a econd time, the subwindow containing it is simply put in the foreground, in order to help the user keeping the central pane as tidy as possible. The function `check_kill_permission` has a control flow graph specific to checking functions: multiple successive branchings where one corresponds to a failure case and is a shortcut to the end of the function, and the other one goes on to the next check. At the end of the function, we find a call to `security_task_kill`, which is the LSM function we were looking for. When we enter it, we see in the history view that we have changed of kernel source file. The function is implemented in `security/security.c`. If we wish to see what are the other functions exported in this file, we can double-click it in the right pane, the source explorer. This updates the "directory" and "file" filters in the left pane. So finally, we have our answer. Guided with previous knowledge of the synchronization mechanisms as well as the high-level documentation of the system call we were exploring, we have traced a possible execution until reaching a predetermined point and the history records our exploration path.

VI. Conclusion and Future Work

We built Kayrebt, a toolset of which we have presented two tools: Extractor, a GCC plugin that extracts activity diagrams out of functions during their compilation and Viewer, a GUI for exploring and navigating through the diagrams produced by Extractor in a codebase. Together, they can be used to help the analyst to understand a large codebase he is not especially familiar with, to get an idea of the control and data flow of a function at a glance, or to identify the static function call chain required to perform an operation. This is useful for codes having several entry points such as an operating system kernel or a GUI.

Many improvements are already under study or being implemented in both Kayrebt::Extractor and Kayrebt::Viewer. In Extractor, the categorization of nodes is made through regular expressions. This is not optimal because this is not related neither to the syntax nor to the semantics of codebase. It would be more useful to have a kind of semantic categorization. With this kind of categorization, it would be possible for example to highlight the locking and unlocking of locks with a different representation for each one. In Viewer, the next step will be to provide users with a mean to save their exploration work for later reuse. This would include opening back the visualizer in the exact state in which it was closed. We also plan to make the tool not only a visualizer but an editor. Example of use cases are annotations made by the user on diagrams. Being able to manually categorize some nodes and apply attributes such as the background color, etc. would be useful.

References

[1] A. Danial, "CLOC – Count Lines of Code," Jul. 2014. [Online]. Available: http://cloc.sourceforge.net/

[2] the Austin Group, "The Open Group Base Specifications Issue 7 IEEE Std 1003.1™, 2013 Edition," 2013. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/

[3] R. M. Stallman and the GCC developer community, "Using the GNU Compiler Collection (GCC)," Tech. Rep., 2013. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.8.4/gcc/

[4] E. Klotz, "Moritz the Nassi-Shneiderman Diagram-Generator for Doxygen." [Online]. Available: http://moritz.sourceforge.net/

[5] "Crystal FLOW." [Online]. Available: http://www.sgvsarc.com/Prods/CFLOW/Crystal_FLOW.htm

[6] D. A. Kosower and J. J. Lopez-Villarejo, "Flowgen: Flowchart-Based Documentation for C++ Codes," *arXiv preprint arXiv:1405.3240*, 2014. [Online]. Available: http://arxiv.org/abs/1405.3240

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: http://portal.acm.org/citation.cfm?doid=115372.115320

[8] The GCC developer community, "Static Single Assignment, internals of the GNU compilers." [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/SSA.html

[9] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software: Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000. [Online]. Available: www.graphviz.org