Generating runtime call graphs

Sebastian Krahmer *krahmer@suse.de*

May 10, 2006

Abstract

This paper describes how function call graphs can be generated during runtime. Solutions for C-programs and Perl scripts are shown.
It is recommended that one use a PDF viewer with high zoom-in capability since some graphs require in-depth zooming.
Keywords: Auditing, call graphs, function call graphs, dynamic graph generation.
search engine tag: SET-krahmer-call-graphs-2005[1]

1 Introduction

Generating call graphs from binary files to understand their execution flows has a history in reverse engineering and binary auditing [2], [9]. Also in the source code auditing world such call graphs can be very helpful. In particular if one is interested into analyzing which part of code is executed with which permissions or whether certain functions are called prior to authentication routines.
The new *-finstrument-functions* switch of GCC allows for easy generation of such call graphs for C-programs. Also scripting languages such as Perl have powerful mechanisms which allow for the generation of call graphs at runtime.

2 Related Work

Much work has already been done to visualize software and its dependencies and structure. However most of the work covers static analyses [9],[11] or describes theoretical approaches how to handle the software's complexity, software maintenance and its graphs [12]. Also for host-based Intrusion Detection Systems the visualization and normalization of software is quite important [11].
Representing running software as call graphs or in some other human-readable or script-readable form seems to be an interesting topic and worth some research.

[1] A preliminary version of this work has been presented in several workshops in 2005.

3 Call Graphs for the C language

There are commercial tools available which generate static call graphs from source code files [3]. For dynamic call graphs, which are generated during runtime, one has to use special compiler features or debugger-like generation tools. The Open Source GNU compiler collection (GCC) [1] offers various compile time features which can be used to generate call graphs at runtime. To achieve this goal, one has to compile the source of the program to be observed with the *-finstrument-functions* switch. This has to be done for every object file. If the program is compiled in this way, GCC places a function-call to `__cyg_profile_func_enter` upon entering and a call to `__cyg_profile_func_exit` upon leaving of every function[2] into the code. An example is shown in Figure 1.

Once the compiler has inserted the instrumentation code, the programmer has to define these functions. This task is very simple. The address of the function which is called is passed as argument to the instrumentation functions. Hence one can easily determine, via a symbol-table-lookup[3], the name of the function and one can build, dynamically, a chain of the functions called and then log this to files. Along with this, appropriate information such as the UID, EUID or the list of open files could be logged.

4 Tools needed

First of all, GCC has to support the instrument functions. Additionally one can obtain a sample implementation for generating call graphs from [4]. This requires the *graphviz* package available from [5] to afford a translation of the `.dot` files into `.pdf` files. The *instrumental* package contains a C-file which has to be modified to reflect the places of the log-files for the `.dot` and the ASCII output. It is then compiled via GCC:

```
user@linux:~> cc -fPIC -c instrumental.c
user@linux:~> ld -Bshareable instrumental.o -o inst.so
```

on ELF32 architectures

```
user@linux:~> cc -m32 -fPIC -c instrumental.c
user@linux:~> ld -melf_i386 -Bshareable instrumental.o -o inst.so
```

on ELF64 (x86_64) architectures. In the second case the program must also be compiled with the *-m32* switch. As an alternative there is an ELF64 version, `instrumental64.c`, shipped with [4] which can be compiled and linked into ELF64 binaries as shown in the first example.
The output of this step is a shared object file which is then just pre-loaded via `LD_PRELOAD` with the program one wants to analyze. The program has to be

[2] Every function called by the program and part of the program. Since libc is usually compiled without that switch, no monitoring is done for libc calls.

[3] Compilation with the *-g* switch is needed.

Figure 1: How function f() is called using the instrument functions.

```
.LC0:
        .string   "f()\n"
        .text
        .align 2
.globl f
        .type     f,@function
f:
        pushl     %ebp
        movl      %esp, %ebp
        pushl     %ebx
        subl      $4, %esp
        subl      $8, %esp
        pushl     4(%ebp)
        pushl     $f
        call      __cyg_profile_func_enter
        addl      $16, %esp

        ...

        pushl     4(%ebp)
        pushl     $f
        call      __cyg_profile_func_exit
        addl      $16, %esp
        movl      %ebx, %eax
        movl      -4(%ebp), %ebx
        leave
        ret
```

compiled with the *-g* and *-finstrument-functions* switches. The glibc contains dummy stubs for the instrument functions so the program even runs without preloading[4]. Alternatively one may link `instrumental.o` statically into the program.

Once the program has been run, the ASCII and `.dot` log-files with the call graph appear at the place specified. Sample graphs generated from the `.dot`-files for an SSH client and server and an HTTP client are shown in Figures 2, 4 and 5. Figure 5 is the ASCII version[5] of the graph in Figure 4 while all the other figures show graphs generated from the `.dot`-files. The ASCII version also contains the caller's UID/EUID and can be *grep*ed more easily for certain call-sequences which need to be found.

The `.dot` file, which is generated, is missing the last closing } since the package does not intercept `exit()` calls or aborts. One has to add it by hand. One can then use the build-pdf script to generate the `.pdf` file.

Some of the graphs such as in Figure 2 can not be viewed in certain PDF viewers (not to speak about printed versions) because larger programs produce complex graphs with a large amount of nodes. For this reason, a zoomed-in version of Figure 2 is shown in Figure 3 just to show that the output is indeed useful. The edges between the nodes are colored differently to make it easier to follow call sequences. They have no special meaning such as clustering or alike. The *graphviz* tool which actually draws the graph from the `.dot` file is responsible for the layout; clusters or sub-graphs may be misleading: the graph produced by the instrumental package is flat.

[4] It does not produce any call graph in this case.

[5] It is truncated to one page since the ASCII version is huge.

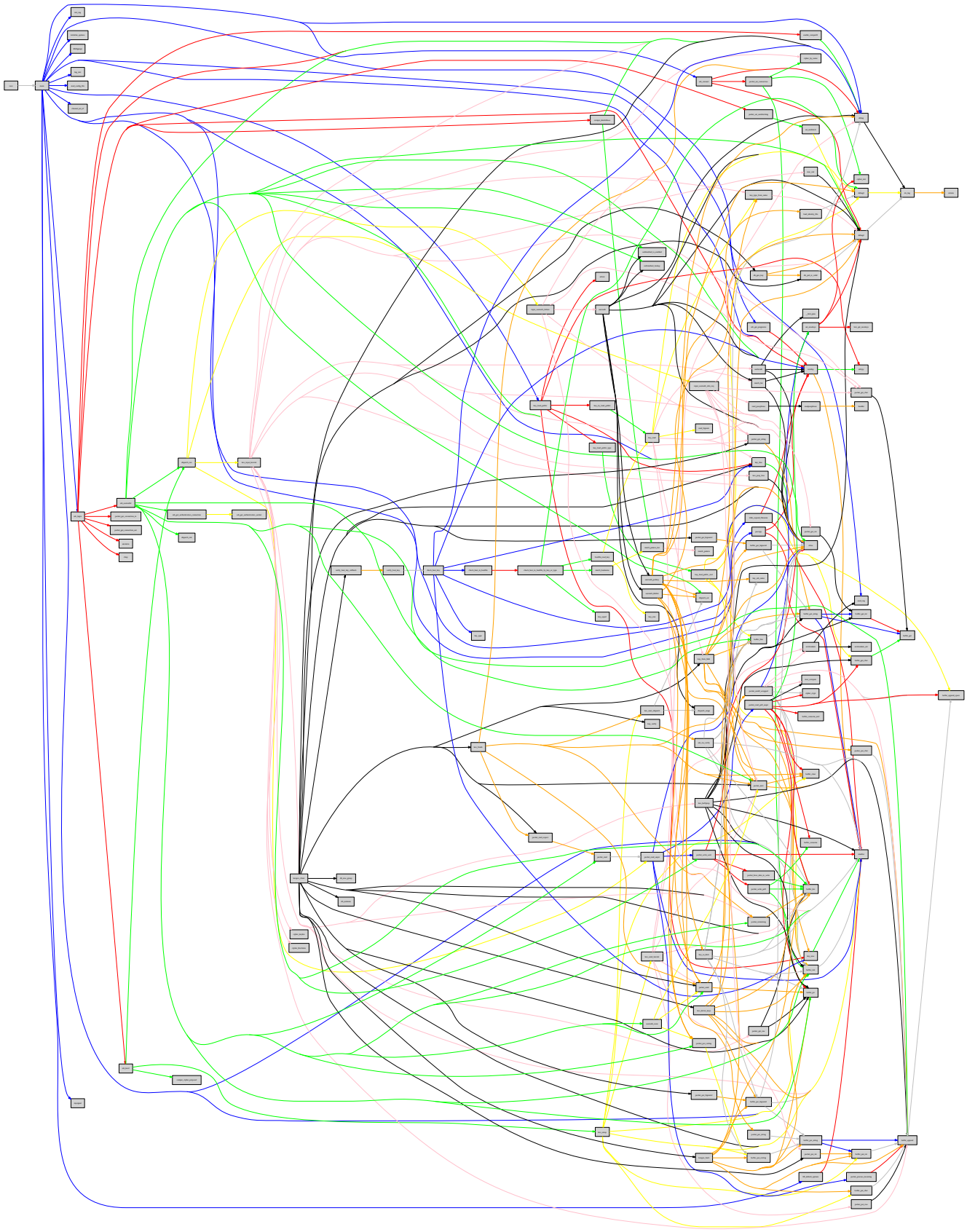

Figure 2: Runtime call-graph of the OpenSSH client during a connect to a server. The different coloring of the edges is only made to allow for easier tracking of calls.

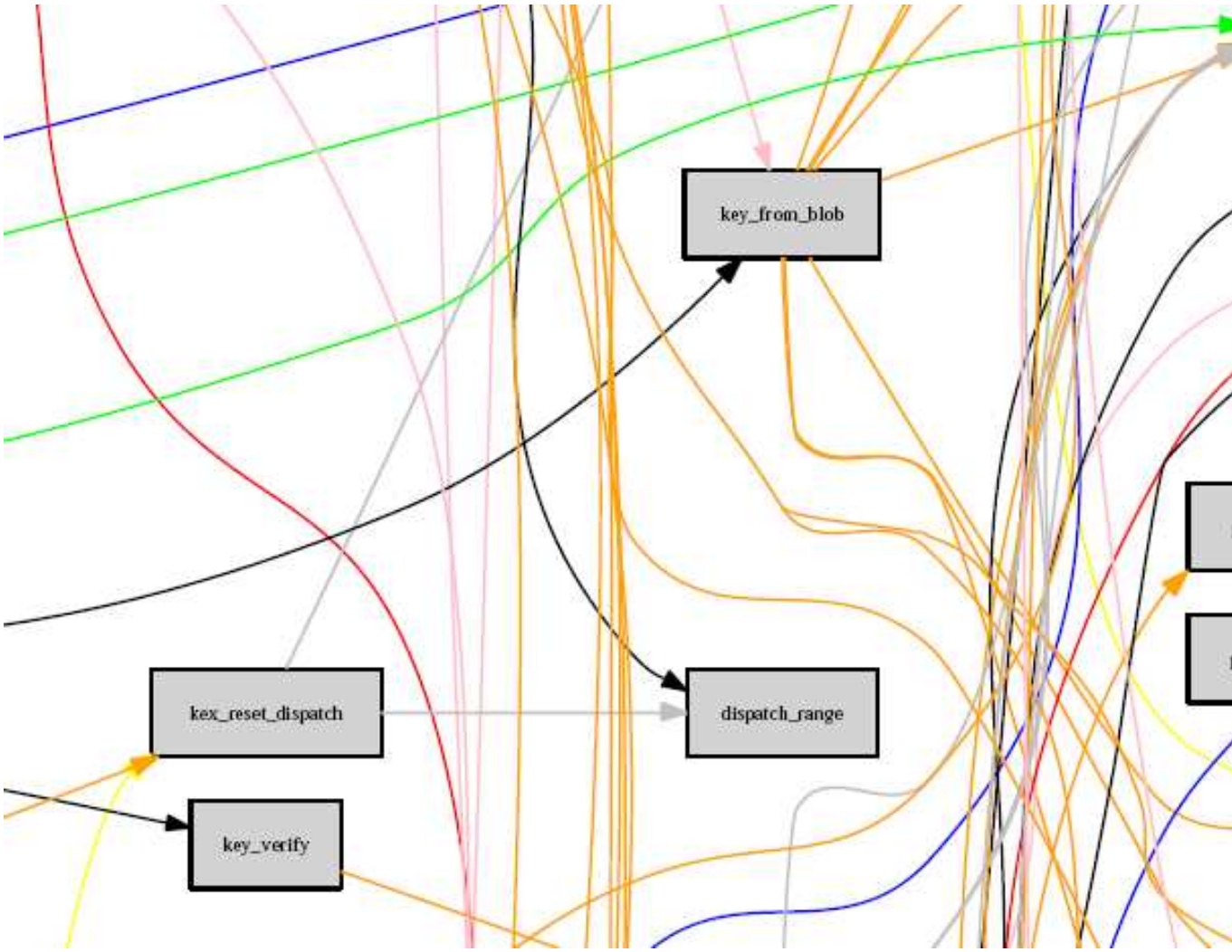

Figure 3: Zoom in of Figure 2.

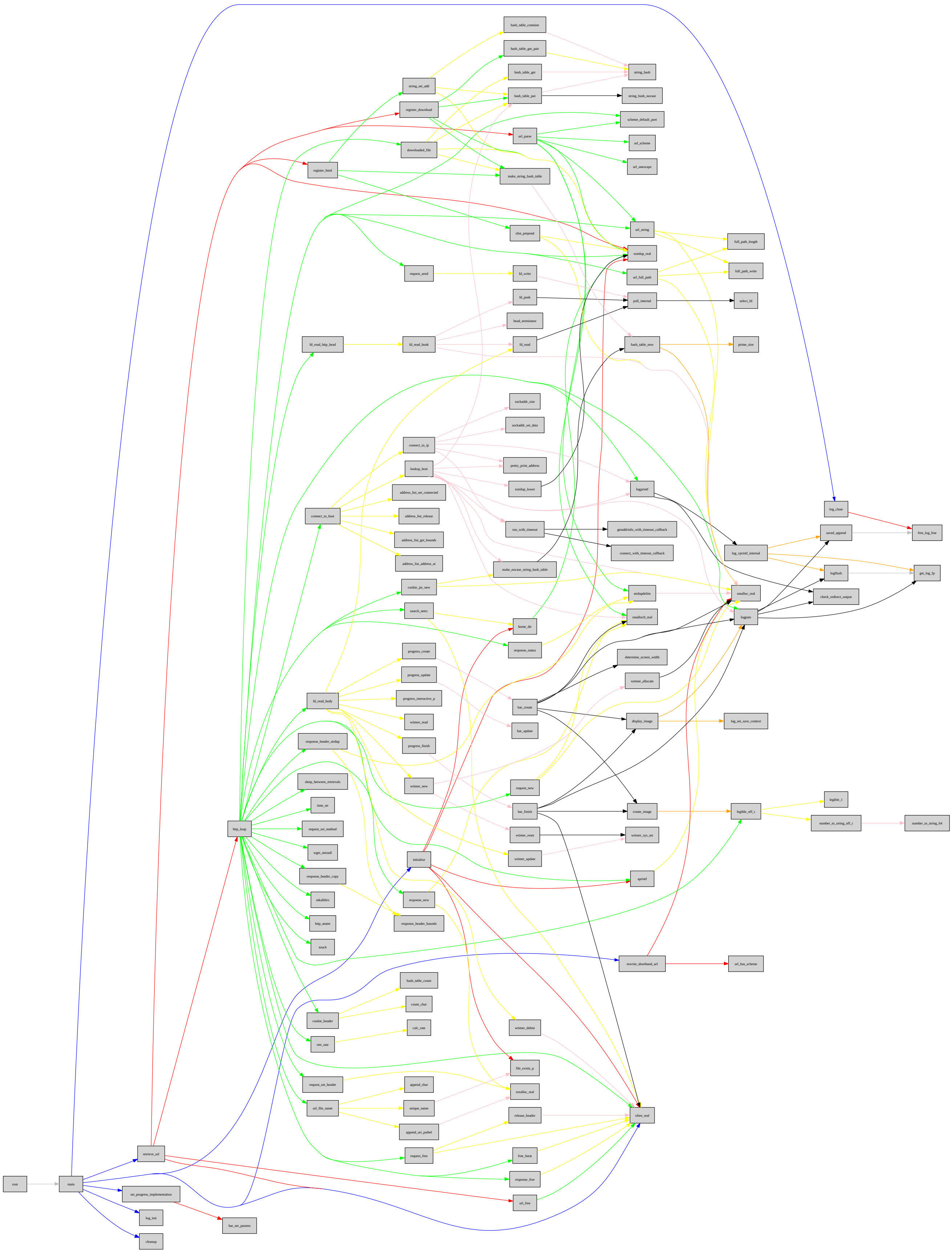

Figure 4: Runtime call-graph of *wget* downloading a HTML file.

Figure 5: ASCII version of the runtime call-graph in Figure 4

```
~ [0000|0000|0000] -- main (0x8063120)
~ [0000|0000|0000]   |-- initialize (0x8062020)
~ [0000|0000|0000]   |  |-- xstrdup_real (0x8072b10)
~ [0000|0000|0000]   |  |-- xstrdup_real (0x8072b10)
~ [0000|0000|0000]   |  |-- file_exists_p (0x80722a0)
~ [0000|0000|0000]   |  |-- home_dir (0x8061e00)
~ [0000|0000|0000]   |  |  |-- xstrdup_real (0x8072b10)
~ [0000|0000|0000]   |  |-- aprintf (0x8072980)
~ [0000|0000|0000]   |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |-- xfree_real (0x8072a20)
~ [0000|0000|0000]   |  |-- file_exists_p (0x80722a0)
~ [0000|0000|0000]   |  |-- xfree_real (0x8072a20)
~ [0000|0000|0000]   |-- set_progress_implementation (0x8065570)
~ [0000|0000|0000]   |  |-- bar_set_params (0x8065700)
~ [0000|0000|0000]   |-- rewrite_shorthand_url (0x806b6f0)
~ [0000|0000|0000]   |  |-- url_has_scheme (0x806a050)
~ [0000|0000|0000]   |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |-- log_init (0x8062520)
~ [0000|0000|0000]   |-- retrieve_url (0x8068b20)
~ [0000|0000|0000]   |  |-- xstrdup_real (0x8072b10)
~ [0000|0000|0000]   |  |-- url_parse (0x806b8f0)
~ [0000|0000|0000]   |  |  |-- url_scheme (0x806b840)
~ [0000|0000|0000]   |  |  |-- scheme_default_port (0x806a120)
~ [0000|0000|0000]   |  |  |-- xmalloc0_real (0x8072c50)
~ [0000|0000|0000]   |  |  |-- strdupdelim (0x8071700)
~ [0000|0000|0000]   |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |-- strdupdelim (0x8071700)
~ [0000|0000|0000]   |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |-- xstrdup_real (0x8072b10)
~ [0000|0000|0000]   |  |  |-- xstrdup_real (0x8072b10)
~ [0000|0000|0000]   |  |  |-- url_unescape (0x8069f50)
~ [0000|0000|0000]   |  |  |-- url_unescape (0x8069f50)
~ [0000|0000|0000]   |  |  |-- url_string (0x806a640)
~ [0000|0000|0000]   |  |  |  |-- full_path_length (0x806a200)
~ [0000|0000|0000]   |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |  |-- full_path_write (0x806a280)
~ [0000|0000|0000]   |  |-- http_loop (0x805f2a0)
~ [0000|0000|0000]   |  |  |-- cookie_jar_new (0x804f4d0)
~ [0000|0000|0000]   |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |  |-- make_nocase_string_hash_table (0x8058a90)
~ [0000|0000|0000]   |  |  |  |  |-- hash_table_new (0x8058980)
~ [0000|0000|0000]   |  |  |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |  |  |  |-- prime_size (0x8058700)
~ [0000|0000|0000]   |  |  |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |-- url_file_name (0x806b270)
~ [0000|0000|0000]   |  |  |  |-- append_uri_pathel (0x806b050)
~ [0000|0000|0000]   |  |  |  |  |-- xrealloc_real (0x8072b80)
~ [0000|0000|0000]   |  |  |  |-- append_char (0x806afd0)
~ [0000|0000|0000]   |  |  |  |-- unique_name (0x8072470)
~ [0000|0000|0000]   |  |  |  |  |-- file_exists_p (0x80722a0)
~ [0000|0000|0000]   |  |  |-- sleep_between_retrievals (0x8068960)
~ [0000|0000|0000]   |  |  |-- time_str (0x80728c0)
~ [0000|0000|0000]   |  |  |-- url_string (0x806a640)
~ [0000|0000|0000]   |  |  |  |-- full_path_length (0x806a200)
~ [0000|0000|0000]   |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |  |-- full_path_write (0x806a280)
~ [0000|0000|0000]   |  |  |-- logprintf (0x8062d60)
~ [0000|0000|0000]   |  |  |  |-- check_redirect_output (0x80625f0)
~ [0000|0000|0000]   |  |  |  |-- log_vprintf_internal (0x8062b60)
~ [0000|0000|0000]   |  |  |  |  |-- get_log_fp (0x80622f0)
~ [0000|0000|0000]   |  |  |  |  |-- saved_append (0x80628b0)
~ [0000|0000|0000]   |  |  |  |  |  |-- free_log_line (0x8062400)
~ [0000|0000|0000]   |  |  |  |  |-- logflush (0x80627d0)
~ [0000|0000|0000]   |  |  |  |  |  |-- get_log_fp (0x80622f0)
~ [0000|0000|0000]   |  |  |-- xfree_real (0x8072a20)
~ [0000|0000|0000]   |  |  |-- gethttp (0x805cfc0)
~ [0000|0000|0000]   |  |  |  |-- xmalloc0_real (0x8072c50)
~ [0000|0000|0000]   |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |  |-- url_full_path (0x806a340)
~ [0000|0000|0000]   |  |  |  |  |-- full_path_length (0x806a200)
~ [0000|0000|0000]   |  |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |  |  |-- full_path_write (0x806a280)
~ [0000|0000|0000]   |  |  |  |-- request_set_header (0x805c370)
~ [0000|0000|0000]   |  |  |  |-- aprintf (0x8072980)
~ [0000|0000|0000]   |  |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |  |-- request_set_header (0x805c370)
~ [0000|0000|0000]   |  |  |  |  |-- xrealloc_real (0x8072b80)
~ [0000|0000|0000]   |  |  |  |-- request_set_header (0x805c370)
~ [0000|0000|0000]   |  |  |  |  |-- xrealloc_real (0x8072b80)
~ [0000|0000|0000]   |  |  |  |-- search_netrc (0x8064bf0)
~ [0000|0000|0000]   |  |  |  |  |-- home_dir (0x8061e00)
~ [0000|0000|0000]   |  |  |  |  |  |-- xstrdup_real (0x8072b10)
~ [0000|0000|0000]   |  |  |  |  |-- xfree_real (0x8072a20)
~ [0000|0000|0000]   |  |  |  |-- scheme_default_port (0x806a120)
~ [0000|0000|0000]   |  |  |  |-- aprintf (0x8072980)
~ [0000|0000|0000]   |  |  |  |  |-- xmalloc_real (0x8072bf0)
~ [0000|0000|0000]   |  |  |  |-- request_set_header (0x805c370)
~ [0000|0000|0000]   |  |  |  |  |-- xrealloc_real (0x8072b80)
~ [0000|0000|0000]   |  |  |  |-- request_set_header (0x805c370)
~ [0000|0000|0000]   |  |  |  |  |-- xrealloc_real (0x8072b80)
~ [0000|0000|0000]   |  |  |  |-- cookie_header (0x804e3c0)
~ [0000|0000|0000]   |  |  |  |  |-- count_char (0x806c560)
~ [0000|0000|0000]   |  |  |  |  |-- hash_table_count (0x8058380)
~ [0000|0000|0000]   |  |  |  |-- request_set_header (0x805c370)
~ [0000|0000|0000]   |  |  |  |-- connect_to_host (0x804bc40)
~ [0000|0000|0000]   |  |  |  |  |-- lookup_host (0x80591e0)
~ [0000|0000|0000]   |  |  |  |  |  |-- logprintf (0x8062d60)
~ [0000|0000|0000]   |  |  |  |  |  |  |-- check_redirect_output (0x80625f0)
~ [0000|0000|0000]   |  |  |  |  |  |  |-- log_vprintf_internal (0x8062b60)
~ [0000|0000|0000]   |  |  |  |  |  |  |  |-- get_log_fp (0x80622f0)
~ [0000|0000|0000]   |  |  |  |  |  |  |  |-- saved_append (0x80628b0)
```

5 Call graphs for scripting languages

Some programs for which a call graph might be useful come, unfortunately, not in the C language but in some scripting language such as Perl [6]. This is, in particular, true for CGI scripts and web administration interfaces such as *Webmin* [7]. Since Perl is a scripting language, it is not possible to produce the call graphs as just described for the C language. On the other hand Perl offers a powerful method of function re-definition at runtime, the AUTOLOAD mechanism. This allows tricks like *autoload.pl*:

```
#!/usr/bin/perl

sub Hello
{
    print "Hello world\n";
}


sub AUTOLOAD
{
    print "You tried to call $AUTOLOAD, lets try Hello() instead.\n";
    return Hello();
}


Hellau();
```

which will produce the following output:

```
linux: > perl autoload.pl
You tried to call main::Hellau, lets try Hello() instead.
Hello world
linux: >
```

Whenever an undefined function is called, the AUTOLOAD subroutine is called instead with the name of the requested function in the $AUTOLOAD variable. To make a call graph from this feature is straight forward. We parse the script, substitute every subroutine definition of `sub X` by `sub CALL_X` and therefore have all calls to X undefined. Then a specially prepared AUTOLOAD subroutine is called which records the call for the graph and calls the original CALL_X function afterwards. Such a graph, generated from the info2html CGI script is shown in Figure 6. Another example, a graph from the Webmin mini-webserver is shown in Figure 7.

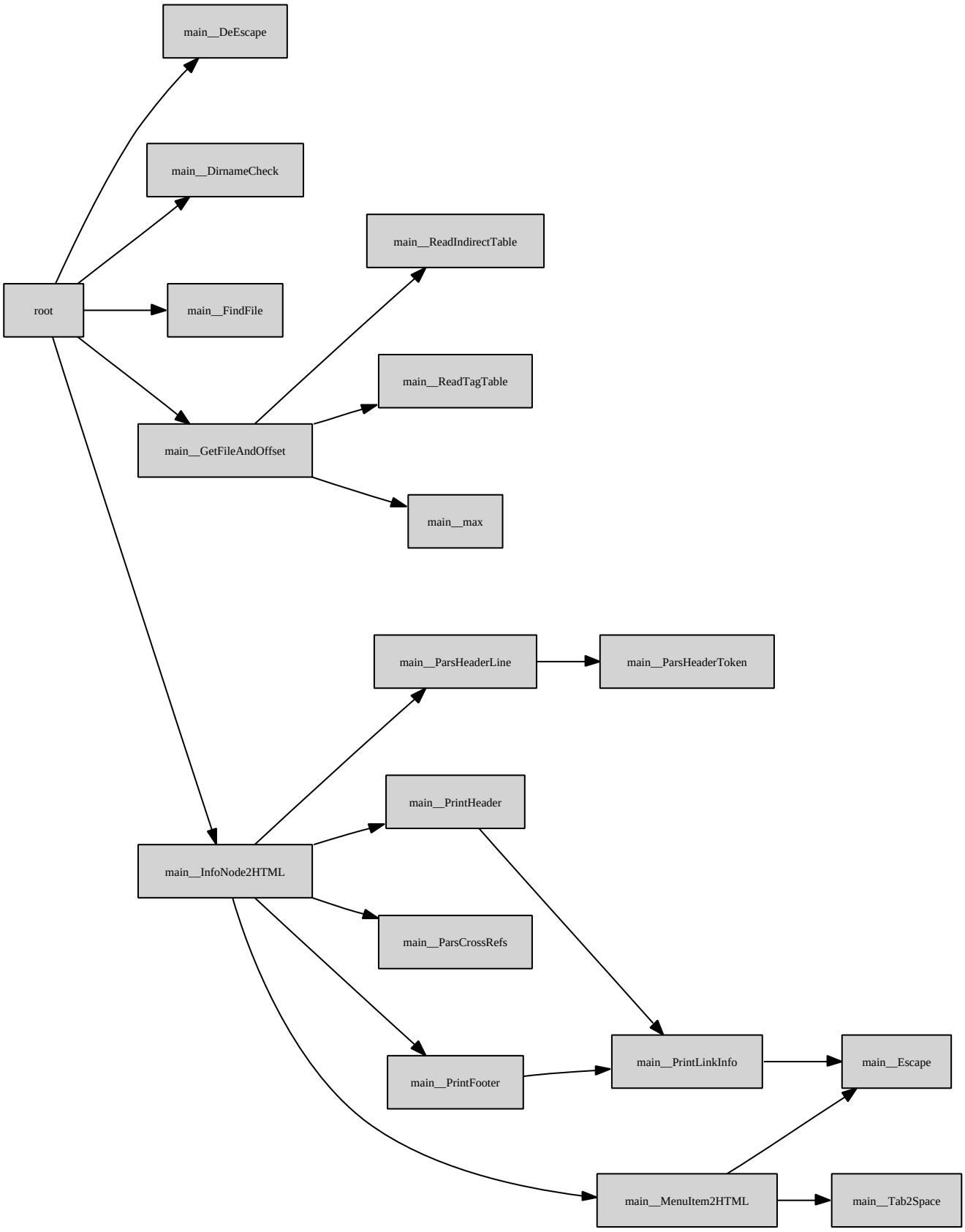

Figure 6: Runtime call-graph for the *info2html* Perl script while it was translating a file.

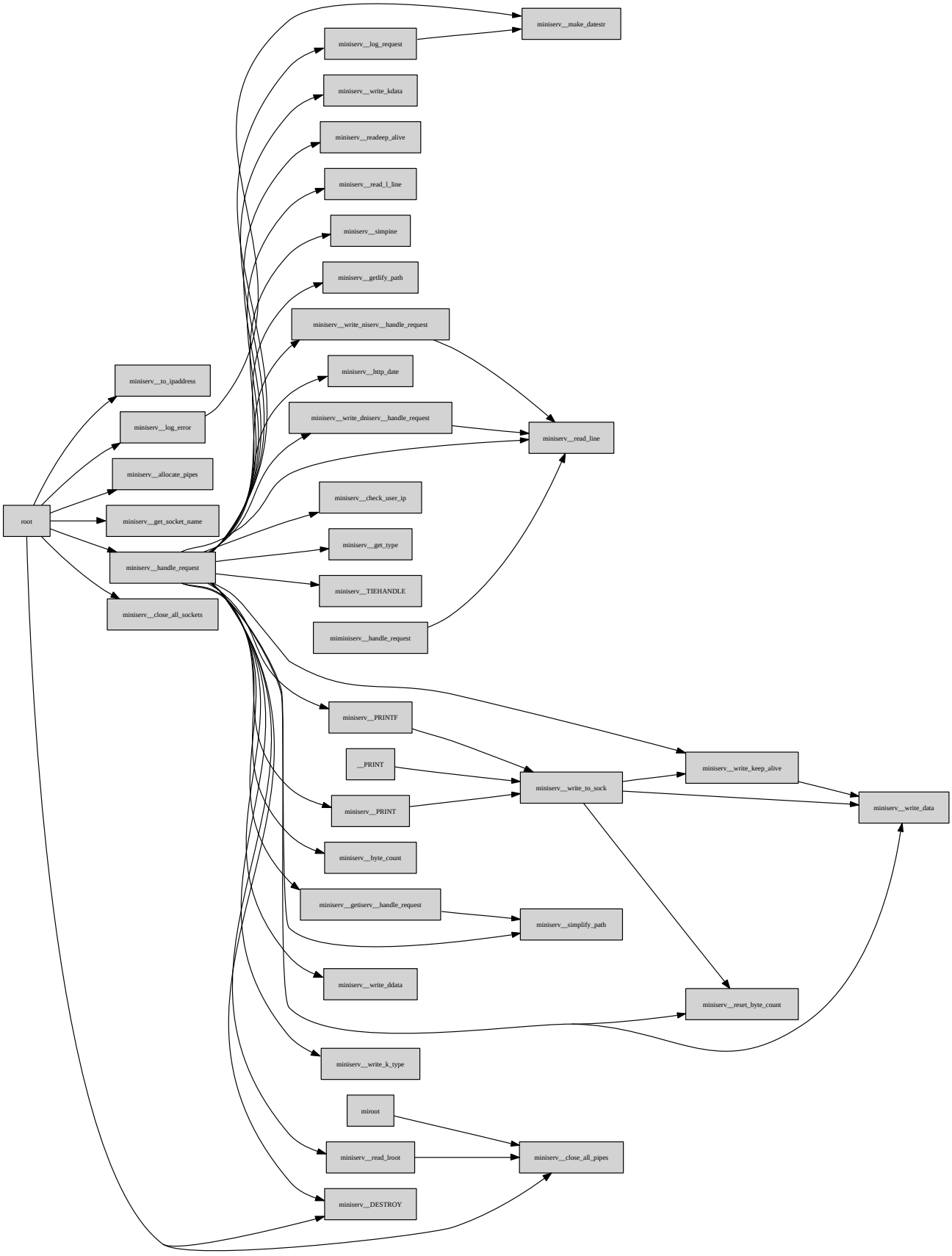

Figure 7: Runtime call-graph of the *Webmin* miniserv.pl Perl script during a admin session.

6 Summary

With [4] one has a good starting point for generating dynamic call graphs. For larger and more complex programs the `.pdf` file will grow and may fast become unreadable. Further research would be needed to split the graphs obtained and make it possible to follow code paths easily. With the ASCII version of the graph however, this is already possible.
The work done in this paper can be understood as the first basic steps towards automatic detection of certain vulnerabilities such as file descriptor leaks or usage of wrong privileges. Although graphs for medium sized or large programs are nearly complete useless for human eyes, tools of mathematics can be used to analyze the program at runtime and maybe to classify certain behavior. Even the input or the data processed by the program at runtime might be recovered by carefully observing the call sequences of the program. Especially the generation of cryptographic keys or large prime numbers (Montgomery Reduction) show patterns which might leak information about input bits.

References

[1] GCC (last access 02-08-2005):

`http://gcc.gnu.org`

[2] Sabre-security (last access 02-08-2005):

`http://sabre-security.com`

[3] aisee (last access 02-08-2005):

`http://aisee.com`

[4] instrumental:

`http://www.suse.de/~krahmer/instrumental`

[5] graphviz (last access 02-08-2005):

`http://www.research.att.com/sw/tools/graphviz`

[6] Perl (last access 02-08-2005):

`http://www.perl.com/`

[7] Webmin (last access 02-08-2005):

`http://www.webmin.com/`

[8] OpenSSH (last access 02-08-2005):

`http://www.openssh.com/`

[9] J.Bergeron, M. Debbabi, M.M.Erhioui, B. Ktari: *Static Analysis of Binary Code to Isolate Malicious Behaviors*, 6 Pages, *LSFM Research Group, Computer Science Department, Science and Engineering Faculty, Laval University Quebec, Canada*

[10] Eric Moretti, Gilles Chanteperdrix, Angel Osorio: *New Algorithms for Control-Flow Graph Structuring*, In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR) 2001*

[11] David Wagner, Drew Dean: *Intrusion Detection via Static Analysis*, In *Proceedings of the IEEE Symposium on Security and Privacy (S&P) 2001*

[12] Shruti Raghavan,Rosanne Rohana,David Leon,AndyPodgurski, Vinay Augustine: *Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases*, In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM) 2004*

A Call graphs and code privilege changes

Although security is not the main topic of this paper or with call graphs in general, they offer an interesting and, in most cases, helpful view of the program.
Especially, system programs written in C are reviewed because they run with super user privileges. Code running as root is potentially dangerous. More complex programs such as OpenSSH [8] use to drop the superuser privileges before they handle untrusted input. This so called *Privilege Separation* mechanism separates the code which handles dangerous input from code which needs to run as root such as looking up user credentials. In case the code handling the input contains a buffer overflow, an attacker can only execute code as unprivileged user.
When generating a function call graph for such a session it makes sense to mark the functions running as root differently from the functions running with user privileges. The *instrumental* package is able to highlight functions (nodes) called with a EUID of 0 in red. Function names are assembled of the pure function name appended by the EUID. If the function `do_authenticate()` is called as root for example, the node name will be `do_authenticate_0`. If two functions are called with different EUIDs there will appear a separate node for each of them in the graph. Sample graphs are shown in Figures 8 and 9.

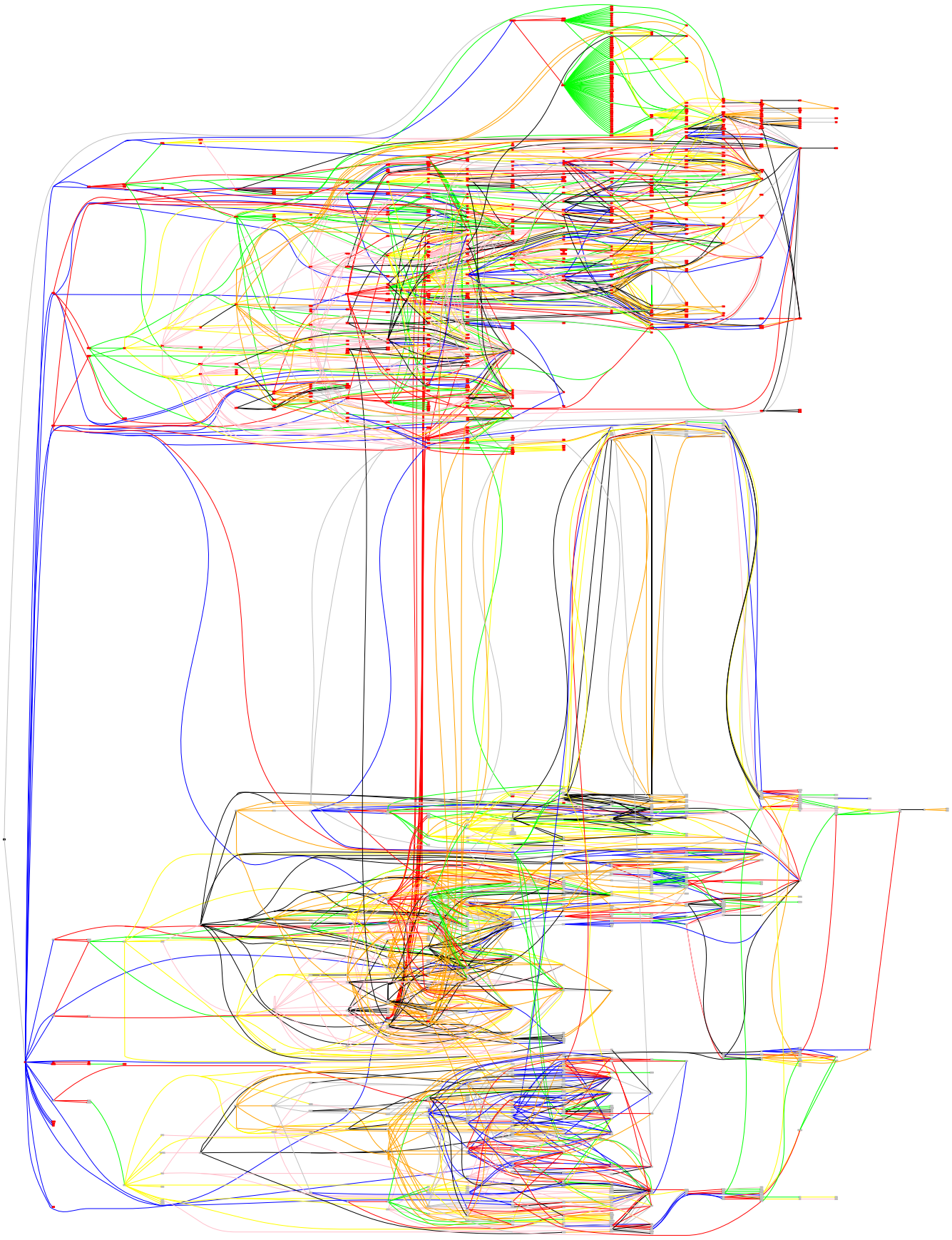

Figure 8: The OpenSSH server including OpenSSL crypto library calls during a client login. Caller EUID is used within the nodes, functions called as root are marked red.

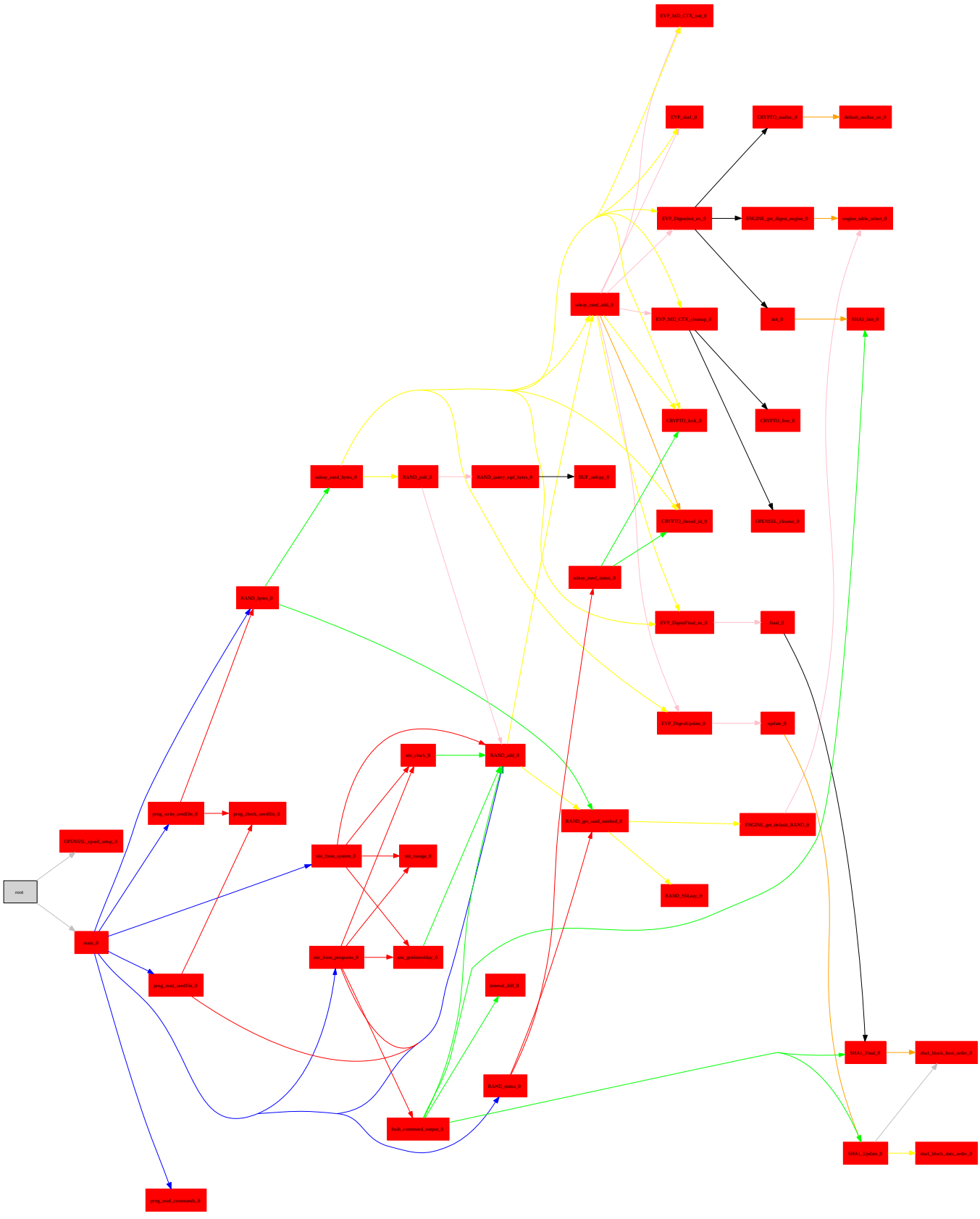

Figure 9: ssh-rand-helper when called during SSH server startup. Again, functions called as root marked red.