KAFÉ: Kernel Analysis Front-End for Software Assurance

Luanne Burns J. Aaron Pendergrass Julian Grizzard
Johns Hopkins University Applied Physics Lab
luanne.burns@jhuapl.edu james.pendergrass@jhuapl.edu julian.grizzard @jhuapl.edu

Abstract

KAFÉ is a tool for kernel inspection, navigation, iterative drill down and analysis with an easy to use interface for searching and sorting on kernel files, functions, types, variables, macros, and symbols. **.** *Although debuggers are geared towards run-time program analysis, the information they rely on offers a powerful basis for the static analysis of programs. The KAFÉ relational database is automatically generated by analyzing the "Debug With Arbitrary Records Format" information. While we gain a great deal of insight into the workings of the Linux kernel and could hopefully recognize potentially problematic violations of data isolation and encapsulation, the scope of the KAFÉ tool goes behind kernel inspection and could be applied to program understanding in general. This paper shows how this approach differs from static source code analysis, run-time analysis or debuggers. The database generation and architecture are described and the interface is illustrated.*

1. Background

Understanding complex computer programs remains a critical challenge facing the software development world. Even relatively simple software developed by a single expert developer quickly approaches a level of complexity where the author is unable to fully explain its behavior. Single author software is exceedingly rare. Most software is developed by large, possibly geographically isolated groups, and must be understood not only by the original developers but also by new team members, quality assurance and testing teams, and numerous others. These problems are nearly as old as the computer program itself, and thus a whole class of software tools exists to aid the software analyst in his attempts to understand a program. Such tools are frequently defined as either static or run-time analyzers.

Static source code analyzers attempt to produce information about a program's behavior by examining the source of the program. Most frequently these analyses use source-code inspection to produce a form of cross referenced report on the lexical structures discovered. The most well known tool in this class is probably *CScope* [12] which was originally developed at Bell Labs for the PDP-11; CS*cope* implements what is described as a "fuzzy parser" for C-like languages; the resultant parse tree is used to guide the creation of a database including symbol names, type specifications, and function call graph data. Another prime example of static source code analysis is the Linux Cross Reference project (*LXR*) [5] which uses similar techniques as *CScope* to generate a database with a conveniently cross-linked web front-end.

Tools based on static source code analysis can be very powerful, but they have some well known drawbacks. Many source code analyzers are unable to fully deal with issues of scoping and namespaces. Problems for source code analyzers also arise when the target source code uses compiler-specific extensions, exercises ambiguous aspects of the language specification, or includes source in a language not directly supported by the analyzer (such as inline assembly). Some tools attempt to overcome these hardships by putting additional burden on the programmer. *Doxygen* [13] is an excellent system which uses source code analysis to create a call graph and interaction diagrams for defined structures; these automatic features can be supplemented by including precisely formatted comments in the source code to document the inputs, outputs, and purpose of each function or type. The disadvantages of source code analysis mostly stem from the fact that writing an effective source code analyzer is akin to writing a compiler where the target architecture is the human consumer. Traditional compiler design targeting computer architecture is hard; the source code analyzer must tackle the same problems of parsing, symbol and type resolution, intermediate processing, and output generation. All of these are nontrivial tasks, and unlike generating machine code, there is no accepted specification for generating output that will effectively convey program meaning to the human consumer. The development of the world- wide-web, hypertext, and database-backed web applications have provided powerful new mechanisms for static, source based analyzers to display their outputs, but even with this arsenal of display mechanisms, the best tools still seem unable to capture more than simple pattern matching over the source code. For a deeper understanding, the analyst is typically forced to sift through a pile of syntactically related though semantically irrelevant hyperlinks, or to go back and read the source code.

Run-time analyzers, on the other hand, are able to provide effective and compelling information on the observable behavior of the software under inspection. Probably the crudest form of runtime analysis (though probably still the most popular) is the use of excessively verbose output during program execution to allow the programmer to compare her own mental model of the intended behavior with the actual computations performed by the program. Though this technique has repeatedly proved its usefulness in debugging program errors, it is comparable to using a large rock to hammer nails: effective, but inefficient. Where the rock fails to exploit the mechanical advantages which led to the development of a hammer, output statements fail to exploit the computational advantages which led to the development of the debugger. Debuggers are marvelous tools which are only slightly less popular amongst programmers than the output statement. Although few programmers are likely familiar with all the features of the average debugger, most are skilled in the use of "breakpoints", the inspection of local and global variable, and manipulating the program stack. Other runtime tools such as IDAPrro [2] profilers and memory checkers are also available and in wide use, but for analyzing cryptic behavior the debugger is still very popular.

Debuggers are powerful tools for the analysis of running programs primarily because of the wealth of information provided by a compiler when building debuggable program images. In standard usage, compilers typically discard a lot of information when building a binary image from source code. For example, type information, variable names, line numbers, and inlined invocations are all irrelevant to the actual execution of the program, and thus are discarded by the compiler. However, when used to generate a debuggable program image, compilers store this information within the executable program file. It is the availability of this information that allows the debugger to provide powerful features for suspending execution on a particular line or function, decoding variables in a running program, listing all active symbols, manipulating the stack, and so on.

Although debuggers are geared towards run-time program analysis, the information they rely on offers a powerful basis for the static analysis of programs. The rest of this paper describes a system which extracts this information from the *Debug With Arbitrary Records Format* [1] sections of a debuggable *Linux Executable and Linking Format* (ELF) [3] file, stores the information in a relational database, and provides a convenient web front-end for interactive browsing and querying of the database. To demonstrate the applicability of this system, it has been applied to the Linux Kernel. We provide examples of interesting statistics gathered from the resultant database, and describe the potential for tools which may act as additional consumers of the database to produce more interesting results. We also provide examples and a comparison with *LXR*. We conclude by discussing some of the shortcomings of our system and comparing its functionality to that provided by other program analysis tools including *DWARF2-XML* [6], that use debugging information as the basis for in depth static program analysis.

2. DWARF Analysis & Database Generation

To aid our ongoing analysis of the Linux Kernel's data structures and flow control, we developed a tool which extracts information contained in the *DWARF* debugging information generated by enabling the "-g" flag to the GNU Compiler Collection (*gcc)* C compiler. We chose to develop such a tool after deciding against the development of a source code analysis tool for a project as complex as the Linux Kernel. Instead of facing the perils of C preprocessing, parsing, inline assembly, and symbol and namespace recognition, we decided to rely on the robust program manipulation powers of the *gcc* toolchain, and reuse the work done by others such as the *GDB* [11] and *libdwarf* [8] in the loading of the far more easily understood *DWARF* debugging information. The tool we developed, which we have dubbed *dwarf2db*, uses preexisting libraries for extraction of the *DWARF* information records, performs some translation and correlation amongst records, and populates a highly interrelated *MYSQL* [10] database with the results of the inspection.

The DWARF Format

DWARF is the widely accepted format for storing debugging information for *ELF* program executables, and has also been ported to the Mach-O binary format used by Apple Inc.'s Darwin/Mac OS X. The *DWARF* format was originally developed with the *ELF* specification by the Unix System Laboratories. Compiling a source file with *DWARF* debugging information enabled adds several debug-specific sections to the resultant object file. These sections are prefixed with ".debug_", and are not mapped into memory when the program is run under normal circumstances (i.e., not in a debugger). The majority of the *DWARF* information is stored in the ".debug_info" section of the object file. Other sections include ".debug_str", ".debug_line", ".debug_loc" which contain ancillary data that is referenced from the ".debug_info" section (specifically those listed contain a string lookup table, line number information, and memory locations of variables respectively).

The ".debug_info" section contains a tree of "Debug Information Entries" (DIEs). Each DIE corresponds to a construct in the original program such as a function definition, a structure or record definition, or a variable declaration. To identify what construct is represented, each DIE is labeled with a tag such as DW_TAG_subprogram, DW_TAG_structure_type, or DW_TAG_variable. In addition to its tag, each DIE has a set of associated attributes. The attributes contain information describing the specific instance of the construct. For example, a DW_TAG_variable may have a name attribute (DW_AT_name), a type specifier (DW_AT_type), and a list of locations at which the variable is stored (DW_AT_locations). The scope of the variable, however, is implicitly defined by its parent DIE. If the variable is local to a subroutine, then it would be a child of the DIE representing that subroutine. Otherwise it would be the child of the compilation unit in which it is declared (a compilation unit corresponds roughly to a '.o' object file). The DWARF format supports references by assigning each DIE a unique ID, thus the DW_AT_type attribute of the variable described above would likely contain a reference to the DIE representing the variable's type. Our tool uses the libdwarf to walk the DIE tree contained in the ".debug_info" section and stores the information contained in the DIE attributes in a relational database. Additionally, the tool examines the TXT section of the object file (the section which contains executable machine code) to extract a complete disassembly of the code, and a partial call graph identifying the source and target functions of all uses of the x86 'call' instruction with a constant target address.

Database Design

The design of the generated database was motivated by our desire to understand the numerous structures and type definitions of the Linux Kernel. Our view is that if we understand the types, and where they are used, we gain a great deal of insight into the workings of the Linux kernel and could hopefully recognize potentially problematic violations of data isolation and encapsulation. With this goal in mind, we divided the space of possible DIE tags into three categories: DIEs concerning datatypes, DIEs concerning variables, and DIEs concerning functions. Each of these major categories is represented as a table in the database, and as a top level browse-able element in the web interface.

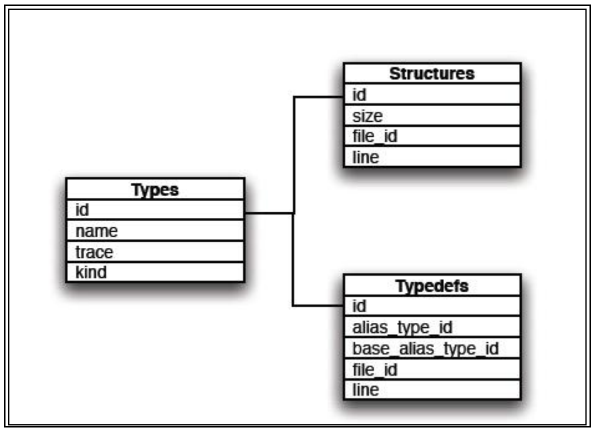


Figure 1. Example Hierarchy Database Schema

However, these categories do not directly map onto the range of DIE types since many different DIE types may all be used to describe data structures, but may have different permissible attributes or children DIEs.

For example, typedef statements are represented by DW_TAG_typedef DIEs which may have attributes indicating the base type, the name of the definition, and the location of the definition in the source code, whereas C struct definitions are represented by DW_TAG_structure_type DIEs which may have similar attributes but do not have a base type, and may contain children DIEs describing the children of the structure being defined. Other DIEs related to datatypes include :
DW_TAG_array_type,
DW_TAG_base_type,
DW_TAG_const_type,
DW_TAG_enumeration_type,
DW_TAG_pointer_type,
DW_TAG_subroutine_type,
DW_TAG_union_type, and
DW_TAG_volatile_type.

Since each of these DIE types describe program constructs with different parameters, each one has a different set of meaningful attributes and children. These are described in detail in [the TIS DWARF-2 format specification]. Similarly, there are numerous DIE types which describe program variables. These include:
DW_TAG_variable,

DW_TAG_formal_parameter, and DW_TAG_member.

To account for this, the database includes a table for each DIE type which represents a specialization of one of the major categories. These tables contain values for the attributes specific to the represented specialization, and share an ID value with an entry in the table for the primary category being specialized. Figure 1 shows an example of this pattern in which the Structures and Typedefs table both provide specializations of the primary Types table. Since both structures and typedefs are associated with a file in which the structure/typedef is declared, both tables contain a file_id field which is used as a key in the Files table, however not all types have such a file (specifically primitive types do not have an associated file) so this field is not a part of the main Types table. The "kind" column of the Types table indicates which table contains the specialization of that row, and the "id" column is used as the key. This system provides a weak form of object-oriented style polymorphic inheritance, and allows the database to concisely store all information pertinent to a given DIE.

The database contains other tables with additional data that does not directly belong to one of the described categories. There are three primary reasons why such a table is included in the design. One reason is to represent attributes whose value is common to many DIE's such as the file in which a variable, datatype, or function is defined or declared. A second reason is to hold multi-part data not suitable for storage in a single table column such as the memory locations at which a particular variable is stored throughout its lifetime. The third reason is to hold information not directly related to any DIE but still of potential value to the analyst such as the call graph and disassembly data generated by direct analysis of the TXT section.

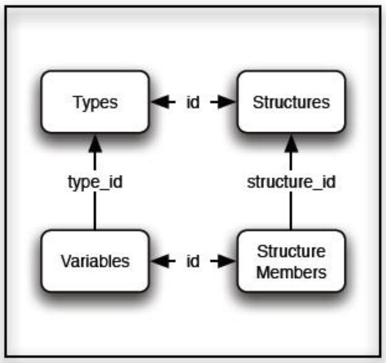


Figure 2a. dwarf2db Relationships

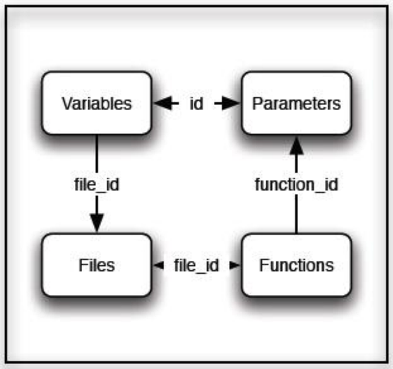


Figure 2b. dwarf2db Relationships

In total the database contains twenty six different tables with 95 columns. Of these columns, fifty six of them are used specifically as keys for multi-table relationships. Figures 2a and 2b depict some of the meaningful relationships modeled within the database. The recognition, storage, and retrieval of these numerous relationships are the source of our system's tremendous potential as an aid to program analysis. The next section describes a web-application which allows user directed browsing and searching of the generated database. Later sections describe the potential for tools which interact either directly with the database, or through the web applications XML interface to implement completely automatic verification of aspects of the Linux Kernel's behavior.

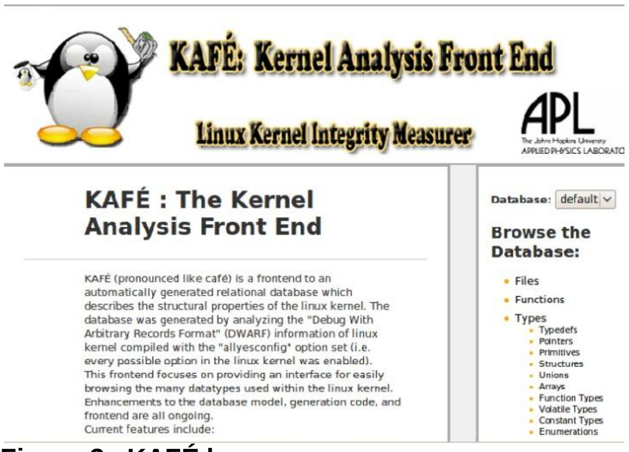
3. Kernel Analysis Front End

Architecture

To exploit the database generated using *dwarf2db*, we developed a web application using the *Ruby-On-Rails* (Ruby-on-Rails) framework to provide a basic Object Relational Mapping. This web application has been dubbed "The Kernel Analysis Frontend" or KAFÉ (pronounced like café) for short. Ruby on Rails was chosen for its reputation as a tool for rapid prototyping of web applications and the extensive documentation available. It should not be construed as the only option for consumption of the generated database. *JavaScript* [4], *CSS* [9] menus, and *PHP* [14] were also employed in the web interface and for communication with *MySQL*.

Interface

The KAFÉ home page gives a brief KAFÉ synopsis and provides a dropdown box from which the user may select a database of interest. After selecting the database, he may choose to start browsing Files, Functions, Types, Variables, Macros, or Symbols. Type and Variables have subcategories, e.g. Types are further broken down into Typedefs, Pointers, Primitives etc while Variables have sub-categories such as Globals and Locals. Figure 3 shows the KAFÉ home page.


KAFÉ: Kernel Analysis Front End
Linux Kernel Integrity Measurer
APL
The Johns Hopkins University
APPLIED PHYSICS LABORATO
Database: default

KAFÉ : The Kernel Analysis Front End

KAFÉ (pronounced like café) is a frontend to an automatically generated relational database which describes the structural properties of the linux kernel. The database was generated by analyzing the "Debug With Arbitrary Records Format" (DWARF) information of linux kernel compiled with the "allyesconfig" option set (i.e. every possible option in the linux kernel was enabled). This frontend focuses on providing an interface for easily browsing the many datatypes used within the linux kernel. Enhancements to the database model, generation code, and frontend are all ongoing.
Current features include:

Browse the Database:

- Files
- Functions
- Types
  - Typedefs
  - Pointers
  - Primitives
  - Structures
  - Unions
  - Arrays
  - Function Types
  - Volatile Types
  - Constant Types
  - Enumerations

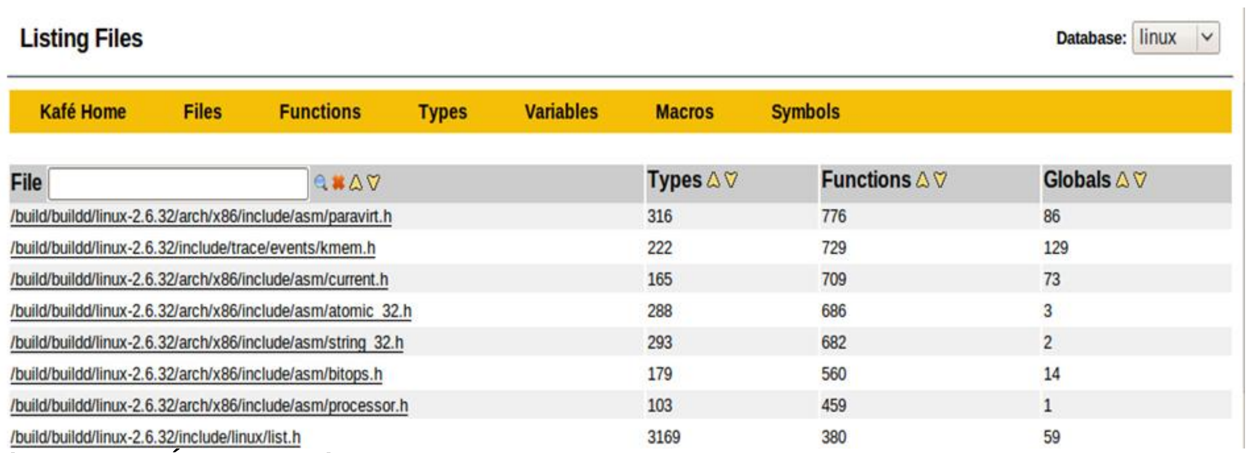
Figure 3. KAFÉ home page

Figure 4 shows the screen displayed after the user selects 'Files' from the options presented. All file names are shown along with a count of their Types, Functions, and Globals. The arrows to the right of the column headers allow easy sorting. The horizontal menu across the top contains the same options presented on the home page for accessing Files, Functions, Types, Variables, Macros, or Symbols as well as returning to the home page. The Database combo box drop down, as on the home page, allows the user to switch databases. The text box next to the file name header allows searching for that column. For all KAFÉ screens, these controls are consistent. If more than one column is searchable, its column header will have a text entry box and the conditions will be ANDED on search. The X next to the text entry box clears the search condition and the magnifying glass launches the search (same as hitting enter).

To better illustrate KAFÉ's features, we will step through an example where our task is to begin exploration in an attempt to find all processes and process ids. Let's suppose that we start by browsing Types from the home page and start our search by looking for *task_struct*. Figure 5 shows the Types listing after KAFÉ has executed our search. From this listing we notice that it is not the *FunctionType* variant that we are interested in but, more specifically, only the *structure* variant so we can further constrain the search, as shown in Figure 6. This page shows us the *task_struct* struct, its size, number of instances, and the file in which it is located. Note that both the file name and the *task_struct* type itself are links which can be clicked on for more information. Clicking on the *task_struct* name produces the screen shown in Figure 7 depicting the struct itself. From here we can drill down further into the struct, link to LXR, or browse to the file in which the struct is defined. The field pid is a member of *task_struct* and is its associated process ID; clicking on it will bring up *more* detail on pid, as shown in Figure 8. This process can be repeated as the user wishes to explore and drill down to deepen program understanding.

Similarly, suppose a user wanted to find the executable name for a task. Once again, a reasonable starting point would be the *task_struct* type as shown in Figure 6. Within the kernel, a process address space, as well as all the information related to it, is kept in an *mm_struct* descriptor so the user can explore this path by clicking on it in the listing. From the *mm_struct* detail, he can drill deeper in his exploration to *exe_file, file, f_path, path, dentry,dentry**, *dname*, and *qstr* detail and finally, to *name*, which is the task name.

KAFÉ can be used for program understanding with programs other than the Linux kernel. Any program compiled with the *allyesconfig* option set can be used by *dwarf2db* to produce an underlying database for the KAFÉ web interface. KAFÉ presents the program, its types, structures, variables and files in a meaningful and browsable format that aids in understanding no matter what the underlying program is.

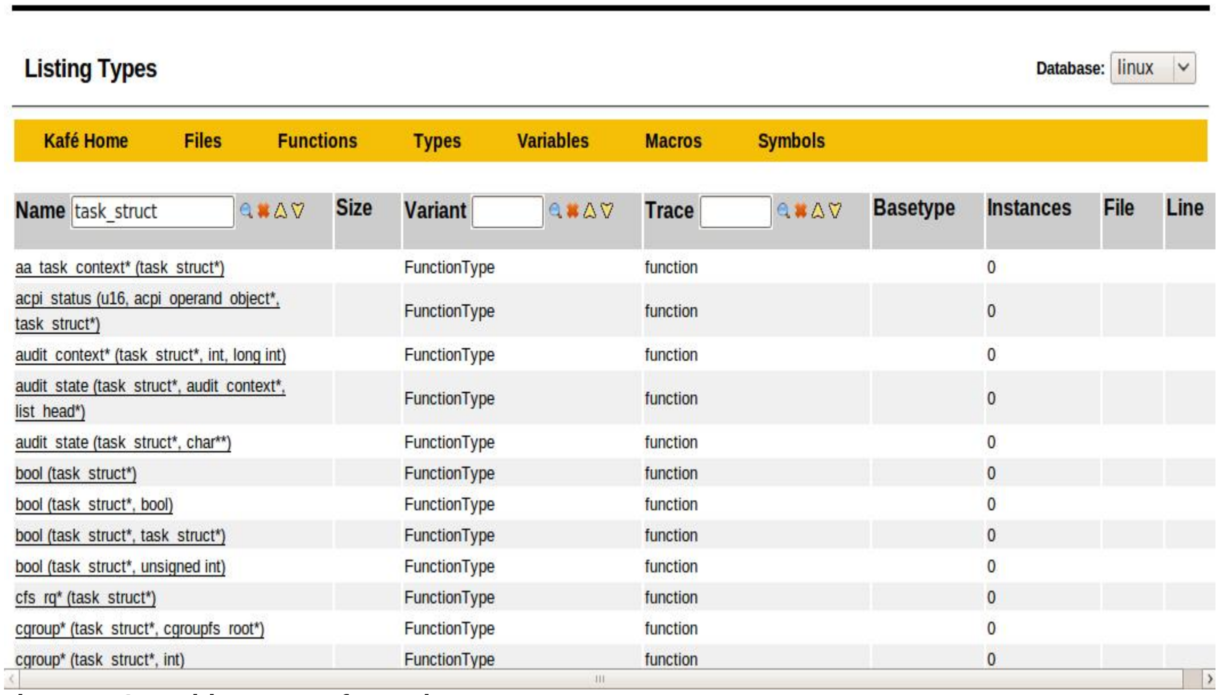


Listing Files

Database: linux

File	Types	Functions	Globals
/build/buildd/linux-2.6.32/arch/x86/include/asm/paravirt.h	316	776	86
/build/buildd/linux-2.6.32/include/trace/events/kmem.h	222	729	129
/build/buildd/linux-2.6.32/arch/x86/include/asm/current.h	165	709	73
/build/buildd/linux-2.6.32/arch/x86/include/asm/atomic_32.h	288	686	3
/build/buildd/linux-2.6.32/arch/x86/include/asm/string_32.h	293	682	2
/build/buildd/linux-2.6.32/arch/x86/include/asm/bitops.h	179	560	14
/build/buildd/linux-2.6.32/arch/x86/include/asm/processor.h	103	459	1
/build/buildd/linux-2.6.32/include/linux/list.h	3169	380	59

Figure 4. KAFÉ menus and controls

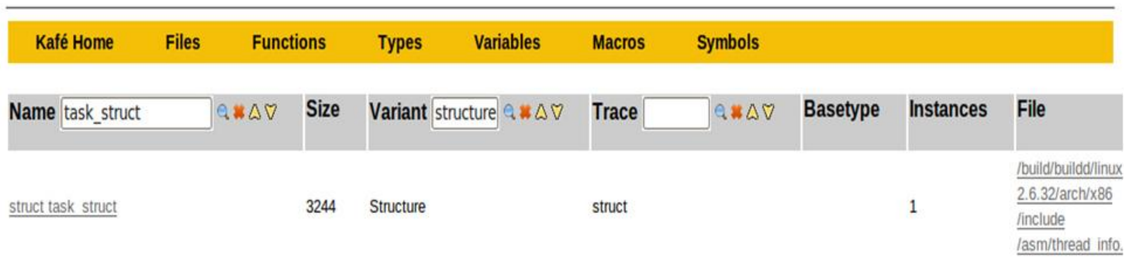


Listing Types

Database: linux

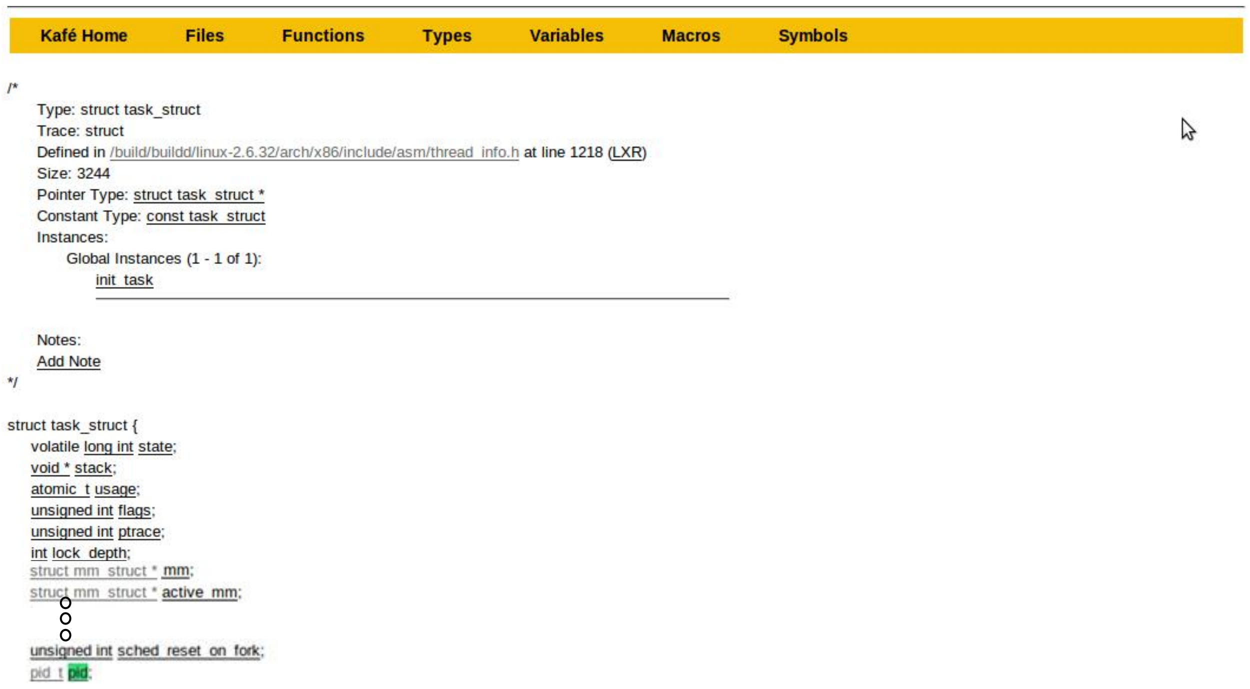
Name task_struct	Size	Variant	Trace	Basetype	Instances	File	Line
aa_task_context* (task_struct*)		FunctionType	function		0		
acpi_status (u16, acpi_operand_object*, task_struct*)		FunctionType	function		0		
audit_context* (task_struct*, int, long int)		FunctionType	function		0		
audit_state (task_struct*, audit_context*, list_head*)		FunctionType	function		0		
audit_state (task_struct*, char**)		FunctionType	function		0		
bool (task_struct*)		FunctionType	function		0		
bool (task_struct*, bool)		FunctionType	function		0		
bool (task_struct*, task_struct*)		FunctionType	function		0		
bool (task_struct*, unsigned int)		FunctionType	function		0		
cfs_rq* (task_struct*)		FunctionType	function		0		
cgroup* (task_struct*, cgroupfs_root*)		FunctionType	function		0		
cgroup* (task_struct*, int)		FunctionType	function		0		

Figure 5. Searching Types for *task_struct*



Name task_struct	Size	Variant structure	Trace	Basetype	Instances	File
struct task_struct	3244	Structure	struct		1	/build/buildd/linux 2.6.32/arch/x86 /include /asm/thread_info.

Figure 6. Constraining search for *task_struct* on Type variant

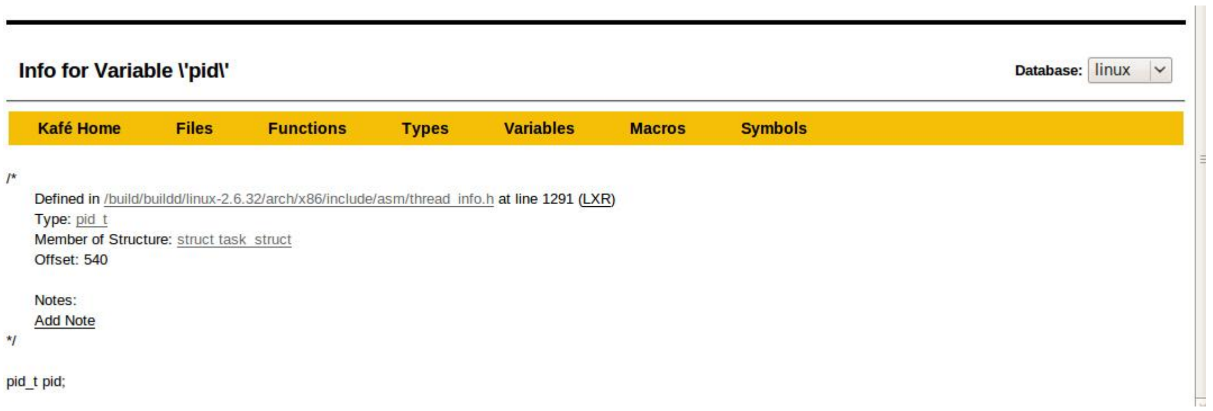


```
/*
    Type: struct task_struct
    Trace: struct
    Defined in /build/buildd/linux-2.6.32/arch/x86/include/asm/thread_info.h at line 1218 (LXR)
    Size: 3244
    Pointer Type: struct task_struct *
    Constant Type: const task_struct
    Instances:
        Global Instances (1 - 1 of 1):
            init_task

    Notes:
    Add Note
*/

struct task_struct {
    volatile long int state;
    void * stack;
    atomic_t usage;
    unsigned int flags;
    unsigned int ptrace;
    int lock_depth;
    struct mm_struct * mm;
    struct mm_struct * active_mm;
    o
    o
    o
    unsigned int sched_reset_on_fork;
    pid_t pid;
```

Figure 7. *task_struct* detail



Info for Variable \'pid\'

```
/*
    Defined in /build/buildd/linux-2.6.32/arch/x86/include/asm/thread_info.h at line 1291 (LXR)
    Type: pid_t
    Member of Structure: struct task_struct
    Offset: 540

    Notes:
    Add Note
*/

pid_t pid;
```

Figure 8. *pid* detail

Comparison to LXR

LXR (formerly "the Linux Cross Referencer") is a software toolset for indexing and presenting source code repositories. It was originally intended for the Linux source code repository but has been used for other code repositories as well. LXR shares some of the same objectives with KAFÉ but differs in a few significant ways.

LXR uses the Linux **source** as its database input while KAFÉ uses a specifically **compiled** Linux kernel with debug symbols as database input. The source that LXR uses could be compiled into a large number of different binaries depending on the configuration while for KAFÉ, the kernel is only meant for a certain class of machines and architecture, e.g. *powerpc* architecture code in binary compiled for 86 architecture. LXR captures all architectures, definitions, *#define*s. KAFÉ removes sections from structures that are #defined out:

e.g. *#ifdef some_disabled_option … #endif“*

LXR includes comments, shows the actual source code for functions and allows searching for types, functions, and free-text, but it is sometimes difficult to specify searches because the configuration is unknown and/or buried in the many displayed and extraneous *#defines*. In KAFÉ, the type space is unique so searches are not ambiguous; additionally size and offset information are available in KAFÉ, LXR cannot display this information because it is not available in the source.

LXR and KAFÉ can be used in conjunction and, in fact, KAFÉ provides hyperlinks to LXR pages to make the synergy more readily available to the analyst. KAFÉ is a tool for understanding the structure of the kernel and allows focusing on one in particular. Sometimes the struct names and member names are insufficient and that's where the source code helps. Note the link to LXR from the KAFÉ screens shown in Figures 7 and 8.

4. Future Work

Aside from these problems in implementation, dwarf2db is constrained by several direct consequences of our refusal to utilize source-code analysis. The most immediately obvious difficulty is that while we are able to gain substantial understanding of the way in which functions and types interact using the DWARF information, our ability to understand the implementation of a particular function is limited to the raw disassembly of the compiled version of the function. Recognizing that source analysis based packages have a dramatic advantage in this particular problem space, KAFÉ includes numerous links to LXR in order to provide a "best of both worlds" type approach.

Related to our inability to capture function behavior is the inability to detect the use of C preprocessor macros. This is an obvious result of not using the source code since preprocessor macros are expanded in the first phase of compilation. Some limited support for describing the available macros may be possible by analyzing the ".debug_macinfo" section. Recognizing the use of macros would require pattern matching the disassembly of functions against the set of possible compiled macros. While this may be possible it is likely to result in many false-positives (pieces of code which are not macro invocations identified as being uses of a macro), and false negatives (uses of macros not being recognized). Also, it would likely take a long time given the large number of macros available in the Linux Kernel, and the number of ntuples of assembly language instructions that would have to be matched against them.

The final two major shortcomings of DWARF based analysis are the inability to penetrate the commonly used abstraction technique of structure embedding, and the difficulty of deducing which variables are used as the arguments to subcalls. The Linux Kernel makes substantial use of embedded structures to provide abstract data types; for example a linked list of some structure type "foo" can be achieved by embedding an element of type "list_head" within the declaration of the structure "foo". The kernel makes available functions including "list_add", "list_delete", and the macro "list_foreach" for manipulating linked lists. If a function is given a "list_head" pointer and would like to recover the "foo" in which the list_head is embedded, the macro

CONTAINER_OF(struct,member,ptr)

is used which operates by subtracting the offset of the structure member from the ptr in and casting it to the structure type. Since these structure embedding rely on macros and behavior implemented entirely within a function, it is extremely difficult to determine e.g., what kind of elements are on a list. The task of deducing the arguments passed to functions is difficult because this information is not recorded directly in the DWARF info, and the argument passing mechanism is compiler specific, and typically varies based on special attributes of the called function. Despite these shortcomings, DWARF analysis in general and dwarf2db in particular offer some features that cannot be easily matched by source based tools.

As described previously, the strength of dwarf2db's approach is its ability to recognize and model relationships between program elements. Because source code analyzers typically have only a limited understanding of scope and namespaces they are often unable to distinguish between different uses of the same identifier. For example, searching for the identifier "inode" in LXR using the 2.6.18 kernel yields 12 structure definitions, 13 variable definitions, and 1234 references (which may be declarations of variables of type struct inode, or references to variables named inode). On the other hand, because the DWARF information includes implicit scoping and namespace information, KAFÉ is able to distinguish between usages of the identifier "inode" and can be used to quickly lookup the type "struct inode" or a particular variable named "inode". Similarly, although most uses of identifiers in LXR source displays are hyperlinks, they link to the results of a lexical search for that identifier. In a large project such as the Linux Kernel, identifier names are often reused so these links can lead to an overwhelming excess of irrelevant information when trying to track the use of a particular variable (e.g., following a link on the identifier "i" from any function that uses this common name for a loop counter). *DWARF* based analysis is capable of directly tracking a particular instance of a variable without being confused by identifier collisions resulting from an inadequate notion of scoping.

Although we arrived at the idea independently, the notion of using *DWARF2* information to produce tools for static program analysis was previously investigated by Gondow, et. al. [7] in describe their development of an XML schema for communicating the information contained in *DWARF2* data. They arrived at many of the same conclusions we did: source based analysis tools rely on difficult and imprecise algorithms, they are typically limited by their inability to understand the deeper semantics of interrelated program concepts (such as variables and their types), and *DWARF* information is relatively standard, easily parsed, and contains substantial relational information that is obscured by source code. They even identified the same technique for developing call graphs from the executable code section, and developed a hybrid system *DWARF*/source system (in much the same vein that we chose to offer links to LXR's source driven engine). Their technique differs from ours primarily in the intended use of their tool; while we focus on developing a system for modeling, visualizing, and consuming the relational aspects of the *DWARF* information, their main focus was on developing a lightweight portable transmission system for *DWARF* in order to enable specialized tools to easily consume the data for their own analyses.

5. Conclusions

KAFÉ is a tool for kernel inspection, navigation, iterative drill down and analysis with an easy to use interface and a backend database allowing searching and sorting on files, functions, types, variables, macros, and symbols. KAFÉ is a front end to an automatically generated relational database generated by analyzing the *dwarf* information. The *dwarf2db* is capable of producing a backend database for any program compiled with the *allyesconfig* option set. Using the database, KAFÉ then provides a web frontend for simplifying program understanding. We have described its use in dynamic analysis and kernel understanding but it could be extended to any program. Its use with other tools such as LXR provides an analyst with a richer set of tools for analysis.

6. References

[1] DWARF, DWARF Debugging Format Standard. http://dwarf.freestandards.org/Home.php, 2006

[2] Eagle, C., The IDA Pro Book. No Starch Press, 2006.

[3] ELF, Executable and linking format. http://www.skyfree.org/linux/references/ELF_Format.pdf.

[4] Flanagan, D., JavaScript: The Definitive Guide, 3rd edition. CA: O'Reilly & Associates, 1998.

[5] Gleditsch, A. G. Linux Cross-Reference. http://lxr.linux.no.

[6] Gondow, K. DWARF2-XML. Japan Advanced Institute of Science and Technology (JAIST), http://www.jaist.ac.jp/˜gondow/ dwarf2-xml/.

[7] Gondow, K. S., Binary-Level Lightweight Data Integration to Develop Program Understanding Tools for Embedded Software in C. Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), 2004, pp. 336-345.

[8] LIBDWARF, DWARF Access Library, Unix International, 1994.

[9] Lie, H. B., Cascading style sheets, WWW Consortium, http://www.w3.org/pub/WWW/TR/WD-cssl, 1996.

[10] MYSQL, MySQL Reference Manual. http://dev.mysql.com/.

[11] Stallman, R. P., Ruby-on-Rails. http://rubyonrails.org, Debugging with GDB. Free Software Foundation. 1994.

[12] Steffen, J. ,The CScope Program: Berkeley UNIX Release 3.2., 1981.

[13] van Heesch, D. Doxygen, http://www.doxygen.org.

[14] Welling, L. A., PHP and MySQL Web Development. SAMS, http://www.php.net, 2001.