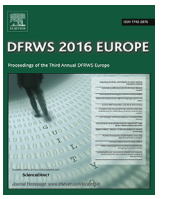
Digital Investigation 16 (2016) S11–S24

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin



DFRWS 2016 Europe — Proceedings of the Third Annual DFRWS Europe

# Automatic profile generation for live Linux Memory analysis

Arkadiusz Socała [a, *], Michael Cohen [b]

[a] University of Warsaw, Krakowskie Przedmieście 26/28, Warsaw, Poland
[b] Google Inc., Brandschenkestrasse 110, Zurich, Switzerland

*Keywords:*
Memory analysis
Incident response
Memory Forensics
Compilers
Reverse Engineering
Malware
Linux Forensics
Digital Forensic Triaging

A B S T R A C T

Live Memory analysis on the Linux platform has traditionally been difficult to perform. Memory analysis requires precise knowledge of struct layout information in memory, usually obtained through debugging symbols generated at compile time. The Linux kernel is however, highly configurable, implying that debugging information is rarely applicable to systems other than the ones that generated it. For incident response applications, obtaining the relevant debugging information is currently a slow and manual process, limiting its usefulness in rapid triaging. We have developed a tool dubbed, the *Layout Expert* which is able to calculate memory layout of critical kernel structures at runtime on the target system without requiring extra tools, such as the compiler tool-chain to be pre-installed. Our approach specifically addresses the need to adapt the generated profile to customized Linux kernels – an important first step towards a general version agnostic system. Our system is completely self sufficient and allows a live analysis tool to operate automatically on the target system. The *layout expert* operates in two phases: First it pre-parses the kernel source code into a *preprocessor AST* (*Pre-AST*) which is trimmed and stored as a data file in the analysis tool's distribution. When running on the target system, the running system configuration is used to resolve the Pre-AST into a C-AST, and combined with a pre-calculated layout model. The result is a running system specific profile with precise struct layout information. We evaluate the effectiveness of the Layout Expert in producing profiles for analysis of two very differently configured kernels. The produced profiles can be used to analyze the live memory through the */proc/kcore* device without resorting to local or remote compilers. We finally consider future applications of this technique, such as memory acquisition.



## Introduction

In recent times, Memory analysis has been used effectively in the wider context of digital forensics, and malware detection (Ligh et al., 2014). In essence, memory analysis strives to make sense of a computer's memory image – an exact copy of the physical memory used by a running system. As the size of physical memory increases, especially on large servers, memory analysis based triaging techniques are becoming more important (Moser and Cohen, 2013).

At first look, physical memory might appear as a large amorphous and unstructured collection of data. In fact, physical memory is used by the running software to store program state in a highly structured manner. The programmer employs logical constructs such as *C structs* to collect related data into logical units, representing abstract data types. The compiler than ensures that this struct is laid out in memory in a consistent way, and generates code to access various members of the struct according to this layout.

* Corresponding author.
*E-mail address:* as277575@students.mimuw.edu.pl (A. Socała).

http://dx.doi.org/10.1016/j.diin.2016.01.004

In order to successfully extract high level information from a memory image, one must extract and interpret the abstract *struct* objects that the program handles from the amorphous physical memory. In order to do this, one must have an accurate model of the physical layout of the structs and their individual member's data types.

Earlier memory analysis solutions relied on hand constructed layout models for each struct, obtained by trial and error (Schuster and Andreas, 2007). However, struct layouts change frequently between released versions (Cohen, 2015b), and the large number of structs of interest makes such maintenance difficult.

For open source operating systems, one might be tempted to examine the source code and from the source code, theorize the precise memory layout for each struct. However (as explained in detail in Section Layout model), such an analysis is not practical without intimate knowledge of the compiler's *layout model*. In practice there are many edge cases which are difficult to predict: For example, the compiler may add padding to ensure alignment of various struct members under different conditions.

In order to support debugging tools, which must also extract meaningful information from the program's memory, compilers typically emit the layout models for each struct used in a program into some kind of debugging stream, for example a PDB file, or DWARF streams (DWARF Debugging Information Format Committee, 2010).

The Volatility Memory analysis Framework (The Volatility Foundation, 2014) was the first open source memory analysis framework able to utilize information derived from debugging streams in order to analyze memory images from multiple versions of an operating system. In the Volatility framework, debugging information is converted into a *profile* specific to a particular version of the operating system. These profiles are embedded inside the tool and allow the user to specify which version of the operating system the image originated from.

On Microsoft Windows systems, debugging symbols are stored in external PDB files which may be downloaded from a central *symbol server* on demand (Okolica and Peterson, 2010). The Rekall memory analysis framework (The Rekall Team, 2014) is able to download debugging symbols for unknown kernels directly from the Microsoft debugging server. This feature is useful when operating in live mode since Rekall can parse the PDB files directly into *profiles* which are used to analyze the running system.

Unfortunately, memory analysis on Linux systems presents some practical challenges. Unlike Windows, the Linux Kernel is typically not compiled with debugging information (such as DWARF streams), nor is debugging information typically available on demand from a debug server. In order to obtain debugging information, one must recompile the kernel, or some part of the kernel (e.g. a kernel module) specifically with debug flags enabled. On a Debian based system, this also requires that a *linux-header* package be installed, containing kernel header files as well as important files that were generated during the kernel compilation step (e.g. *Modules.symvers* file) before a kernel module can be built (Hertzog and Mas, 2014). In practice, the kernel-header package for a custom compiled kernel is often not available or was never even created in the first place. At best, incident responders must scramble to identify the correct kernel-header package for the running kernel on the target system and hope that it matches.

Another complication is the high level of configurability of the Linux kernel. During the kernel build process, users may specify a large number of *configuration options* through the kernel's configuration system. These options affect the kernel build process by defining a large number of C preprocessing macros.

The Linux kernel source uses preprocessing macros heavily to customize the operation of the kernel itself – and in particular the kernel tends to include certain fields into critical structs only if certain functionality is enabled by the user. For example consider the code in Fig. 1 which shows the definition of *task_struct* – a critical struct maintaining information about running processes.

As can be seen, some of the struct members are only included if certain configuration parameters are set. For example, the *sched_task_group* pointer only exists when the kernel is compiled with support for task group scheduling – an optional feature of the Linux kernel. Similarly *CONFIG_SMP* controls the inclusion of several fields used by multiprocessing systems.

When the compiler generates the abstract struct layout model, it must allocate a position for every struct member in memory, sufficient to accommodate the size of the member, its alignment requirements and the alignment of members around it. Clearly if certain fields are not included in the struct definition (e.g. if the feature they implement is not chosen by the user), the compiler will not reserve any space for them, and therefore struct members that appear later in the struct definition will be located at different positions in memory.

The main problem that memory analysis tools encounter when parsing the Linux kernel's memory, is that the configuration of the kernel controls the resulting kernel structures' layout model, but this configuration is not constant. Since Linux users and distributions are free to reconfigure and recompile their kernels at any time, each specific kernel used in a given memory image can have vastly different configuration and therefore layouts (This is contrasted with commercial operating systems, such as Windows or OSX, where only a small number of officially released versions are found in the wild).

One solution to this problem is to maintain a large repository of common kernel configurations. The *Secondlook* product (Secondlook, 2015) maintain a large repository of profiles for every release of major distributions (e.g. Ubuntu, Redhat etc). Although this repository is large (supposedly over 14,000 profiles), if the user has recompiled the kernel and changed some configuration options themselves, the correct profile will not be found in the repository. A complete repository will have to account for every combination of configuration options and would therefore be impractically large.

A different approach, as taken by some memory analysis frameworks (The Rekall Team, 2014; The Volatility Foundation, 2014) requires the user to specifically build a profile for each target kernel in advance prior to analysis. The usual procedure is to obtain the kernel-headers package and use the target kernel's configuration to compile a

```
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;

#ifdef CONFIG_SMP
        struct llist_node wake_entry;
        int on_cpu;
        struct task_struct *last_wakee;
        unsigned long wakee_flips;
        unsigned long wakee_flip_decay_ts;

        int wake_cpu;
#endif
        int on_rq;

        int prio, static_prio, normal_prio;
        unsigned int rt_priority;
        const struct sched_class *sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
        struct task_group *sched_task_group;
#endif

#ifdef CONFIG_PREEMPT_NOTIFIERS
        /* list of struct preempt_notifier: */
        struct hlist_head preempt_notifiers;
#endif

#ifdef CONFIG_BLK_DEV_IO_TRACE
        unsigned int btrace_seq;
#endif
        ...
```

**Fig. 1.** A sample struct definition from the Linux Kernel source. Note that many fields in the struct are conditional on the kernel configuration parameters.

test kernel module with debugging symbols enabled. The compiler will emit the required debugging symbols into DWARF streams in the test kernel module. Memory forensic tools then parse the required DWARF information to construct a suitable profile to analyze the specific kernel's memory.

In an incident response and live analysis context, compiling a kernel module may not always be possible; Servers rarely have a complete compiler tool-chain installed, or the required kernel source code or headers. Conversely, installing development tools on production systems is not always possible. If a profile can not be built locally on the target system, the analyst must install the relevant kernel headers (if they exist) on a similar different system so a profile could be generated. The analyst must then manually copy the generated profile back onto the target system for live analysis.

This extra work makes it difficult to implement a fully automatic memory triaging system (Moser and Cohen, 2013). An automated system can not have manual interaction to construct the profile on every system being analyzed. For example, the GRR Rapid response tool (Cohen et al., 2011) is able to conduct sophisticated memory analysis on a large number of enterprise Windows and OSX systems. However, for Linux, it requires a pre-calculated profile for every possible kernel it might encounter.

Ideally, a profile can be generated automatically purely using information readily available on the running system itself, without prior preparations.

The literature documents a number of efforts to achieve this goal (Lin et al., 2011). For example, *RAMPARSER* (Case et al., 2010) uses dynamic reverse engineering techniques to reconstruct a restricted model of struct layouts (Containing a small selected subset of key struct's fields). For each struct, a reduced set of important fields are reconstructed by reverse engineering exported kernel functions which manipulate these fields. Case et al. (2010) document some of the challenges encountered using this approach – For example, it was noted that the assembler code generated for each function can vary significantly, even within the same kernel version. Therefore the tool must be widely tested to allow for all possible variations of assembler code encountered in the field. Furthermore, each struct member must be examined manually so the correct strategy for deducing it's offset from certain kernel functions be devised. This manual treatment of struct members limit the tool's scalability due to the significant effort required to build detailed model for each struct member.

Cohen (2015b) has extended this idea by developing a disassembler signature template specification. The templates allows for expressing disassembler matching rules to be expressed concisely, while allowing for common

variations such as differences in exact registers used in the assembler code. This system still fails when the disassembly varies more radically, though.

The main limitation with both these approaches is the manual analysis required to identify the assembly patterns required for extracting just a few struct offsets. As modern memory analysis techniques extend their coverage of kernel data structures, more and more struct members are becoming relevant, and manually devising techniques to derive them at runtime simply does not scale.

The variability in kernel struct layouts can be attributed to two orthogonal dimensions. One type of variability is introduced by varying kernel versions and applied patches, while another type of variability is introduced by users recompiling a kernel at a specific release version using different configuration options. This paper specifically deals with the variability introduced due to configuration changes. Ideally an automated solution would be able to handle both types of variability, but dealing with recompiled kernels is an important step towards this final solution.

## Problem statement

Our goal is to improve the specific use case of live analysis in incident response. Ideally we would run our live memory analysis tool from read only external media (e.g. DVD Rom or a read only mounted USB drive) and it would have everything needed to triage an arbitrary Linux system. Such a use case is already common for Windows systems Cohen (2015a), and our goal is to make it just as automated for Linux systems.

We assume the following constraints:

- It is not practical for the user to build a specific tailored profile for the running kernel on the live system itself.
- The specific target kernel version is known (e.g. via the *uname* command).
- The configuration of the target kernel is known and accurate. This is often found in the */boot/* partition. For example */boot/config-3.13.0-61-generic* corresponds to the running kernel with version *3.13.0-61-generic*.
- The *System.map* file is present and accurate (or */proc/kallsyms* is available).

The main difficulty with the current "compile profile locally" approach is that it requires the entire compiler tool-chain to be available on the target system, including the correct kernel headers package. Much of the time, however, responders to production systems are unable to install additional software on the target system, especially those who are suspected of being compromised.

One possible solution is to remove the compilation step from the target server, and perform it on a remote server instead. Currently, responders wanting to perform live triaging, need to manually copy the required files to a remote system, similarly configured to the target system, create the profile file and then copy the profile back to the target system prior to analysis. This approach does not scale, in particular when automated remote live analysis is required for rapid triage and response (as is the case with for example the GRR rapid response tool (Various, 2015a, 2015b)).

Consider the solution depicted in Fig. 2, which contains the git checkout of the kernel source tree. The server can offer a build service accepting a configuration file, then building the required debug module, and finally serving the profile to the memory analysis tool.

This arrangement is essentially an automated version of the manual profile creation by module compilation approach: The build server uses a local source repository to build a kernel module with debug symbols using the provided configuration. The struct layout is then extracted from DWARF debug streams and returned to the triaged system which completes the profile. The main benefit with such a system is that the compiler tool-chain is not installed on the analysis target itself – rather it is installed on the server, and being used remotely.

However, this design still has the major shortcoming that network access must be maintained with the profile server, which must be available before each new live system can be analyzed. In particular this approach is not convenient from a remote live forensic perspective, since it requires analysis to be paused while the build server generates the profile. Additionally the kernel headers package is still required to be available on the server before it can build a kernel module (Hertzog and Mas, 2014). For custom compiled kernels, the relevant kernel header package must somehow be located and transferred to the build server too.

## Layout expert

It would be much more convenient to contain all required information within the live analysis tool itself. This way, the live analysis tool becomes stand alone, and able to calculate its own profiles without external dependencies.

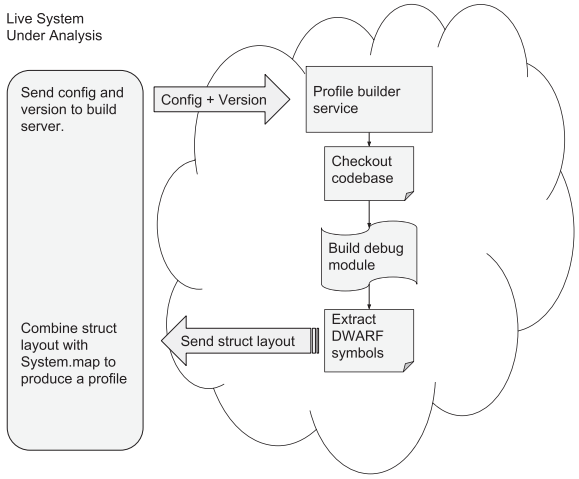



**Fig. 2.** A possible architecture for a profile builder service. A memory analysis system sends the server a configuration file and a kernel version, and the server builds a test module with debug symbols, sending back the struct layout.

We have developed a tool, dubbed the *Layout Expert* which takes the profile generation service to another level. Rather than build a test module using the complete compiler tool-chain, the Layout Expert attempts to predict struct layout by emulating the main operation of the compiler tool-chain locally.

This allows the layout expert to calculate a profile without contacting an external server and without installing the full compiler tool-chain on the target system.

The Layout Expert's main components are illustrated in Fig. 3. The kernel's sources are parsed into a *Pre-AST* (i.e. a C-Preprocessor's Abstract Syntax Tree), which is then trimmed to remove much of the unneeded code, unneeded struct definitions and macro expansions. The *Trimmed Pre-AST* is serialized using JSON to a file and stored on disk. This file can be delivered as part of the data files included in the live memory acquisition tool.

When analysts work with a live Linux system, they apply the target kernel configuration file to the *Pre-AST* to derive a *Preprocessed Pre-AST*. This step essentially reduces the AST to a single string of preprocessed C Code. The pure C code is then trimmed into a *snippets* containing only relevant C code for our purposes. The C code snippets are then parsed by a C parser to produce a *C-AST*.

Finally we apply a layout model, specifically developed for GCC, to the C-AST and derive layout information for all the defined structs. This layout information is combined with the *System.map* to produce a working profile for analysis frameworks.

Our proof of concept implementation of the Layout Expert targets Linux systems running on the AMD64 architecture.

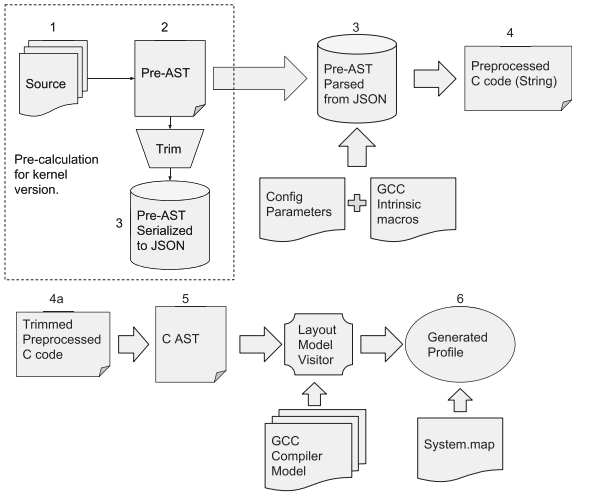



**Fig. 3.** The Layout Expert is divided into three major components: The *Preprocessing parser* constructs a trimmed *Pre-AST* from source code. This is then serialized into JSON and stored locally in the live memory analysis tool. At triage time, the live system configuration is applied to this *Pre-AST* to produce a pure C file (free from macros). As an optimization, the pure C file is further trimmed to only include relevant structs. Finally, the trimmed C file is parsed by the *C Parser* into a C-AST, suitable for application of the *layout model* in order to predict the final layout of each struct and therefore derive a profile.

We now describe each of the processing steps in more detail.

### *The Preprocessing parser*

The C programming language is not a single language – rather it relies on two separate passes over the C source code. The first pass is termed *preprocessing*, and it is typically performed by the C preprocessor (Kernighan et al., 1988).

Preprocessing transforms the source code into pure C code, by use of macro substitution (*#define* directives), file includes (*#include* directives) and conditional includes (*#ifdef* directives).

The Linux kernel makes heavy use of preprocessing directives. For example:

- Different files are included based on CONFIG parameters.

• Macros are expanded in the preprocessing step which affect struct layout, such as the GCC __attribute__(packed) expression.

- CONFIG parameters are evaluated in the pre-processing stage to add or remove members from structs. This effectively changes the field layout of structs depending on config parameters.

Some example snippets of code for such preprocessing examples are shown in Fig. 4. In the first case alternate implementations of a kernel subsystem are chosen based on a configuration choice. These implementations are present in different include files, and each include file may introduce a different set of structs and types specific to that implementation, as well as introduce new macros that affect subsequent processing. In the second case, a macro is defined, that when expanded, adds an attribute to a struct field. As explained in Section Layout model, attributes can influence the struct's field layout and so it is critical for the parser to correctly expand these macros.

From these examples it is clear that in order to properly parse struct layouts, the source code must be preprocessed in a similar way to the C preprocessor. However, while the C preprocessor needs to have the full configuration parameters during pre-processing time, the layout expert's *C Preprocessor* parser does not.

The preprocessor creates a Preprocessor Abstract Syntax Tree (*Pre-AST*) (Figs. 3–2). The *Pre-AST* does not require the configuration parameters at this stage – for example, if there is a conditional include, both conditions are present in the *Pre-AST*. It is only in subsequent processing steps, when CONFIG parameters are known, that we are able to resolve the *Pre-AST* by trimming parts which are selected by the relevant CONFIG parameters. It is worth noting here that the *Pre-AST* contains *TEXT* nodes which contain verbatim snippets of C code. By evaluating the decision nodes (e.g. #ifdef) the relevant text nodes may be glued in the correct order, to produce the complete C code. Note too that text within these *TEXT* nodes must be expanded by the

include/linux/rcupdate.h (line 399)

```
#if defined(CONFIG_TREE_RCU) || defined(CONFIG_PREEMPT_RCU)
#include <linux/rcutree.h>
#elif defined(CONFIG_TINY_RCU)
#include <linux/rcutiny.h>
#else
#error "Unknown RCU implementation specified to kernel configuration"
#endif
```

include/linux/compat.h (line 146)

```
# define __packed               __attribute__((__packed__))

struct compat_sigaction {
  ...
  compat_sigset_t     sa_mask __packed;
};
```

**Fig. 4.** Some examples of the use of C preprocessing in the Linux kernel. The first example illustrates conditional header includes based on configuration parameters. The different files included may declare different structs or struct layouts for the same type. In the second example, we see a macro expansion which influences the layout policy for a specific field by specifying the packing attribute on that field (See section Layout model).

preprocessor emulator in order to substitute macros (e.g. as used in Fig. 4).

An example of a JSON encoded *Pre-AST* tree is shown in Fig. 5.

*Trimming the Pre-AST*

As described previously, the Layout Expert's preprocessor converts the kernel's source files into the *Pre-AST*. The *Pre-AST* is then serialized and incorporated into the live memory analysis tool. However, this step is effectively the same as simply including the source files directly within the analysis tool. In fact the size of the serialized JSON data exceeds that of the plain source files due to the extra information stored in the AST.

The main benefit comes when we consider that in order to analyze the memory of the running system, we do not require the entirety of the kernel source. We are typically only interested in a small subset of the structs defined and used by the kernel. We certainly do not need any of the source code itself.

The Layout Expert's *Pre-AST Trimmer* reduces the size of the *Pre-AST* in such as way that it does not impact subsequent processing. By identifying and removing branches of the *Pre-AST* which are irrelevant to the final profile, we can reduce its serialized size.

The challenge is to identify and remove unnecessary code from the Pre-AST, while still maintaining all the needed code – either the actual structs we need or their dependencies.

We have come up with several strategies:

- Develop a subset of struct definitions which are actually used by our memory analysis framework.
- If we restrict our tool to only support certain configuration values (e.g. kernel architecture), we can partially preprocess the Pre-AST and remove large chunks of unneeded code.
- Remove inline function definitions since we do not generate any code. We are only interested in struct definitions.

Our efforts have resulted in a significant reduction of total *Pre-AST* size already, but we believe that further research in this area can reduce the *Pre-AST* size even more. Currently the trimmed Pre-AST is around 6 mb uncompressed and around 600 kb compressed with *gzip*.

*Preprocessing the Pre-AST*

On the live system we obtain the kernel's configuration, effectively creating a large number of macro definitions. Armed with this knowledge, and a model of GCC intrinsic macros, we can preprocess the *Pre-AST* into a pure C file (i.e. free from macros and other preprocessing directives). This can be seen in Fig. 3 part 4.

*Second level trimming*

The resulting pure C file is very large (over 50 k lines of code) and defines every single data structure used in the Linux kernel. Since the *Layout Expert* is written in Python, where complex parsing is resource intensive, we employ a second trimming phase (Shown in Fig. 3 part 4a) in order to optimize performance. In this phase we use a rough parser to quickly partition the C file into *snippets*. Each snippet describes the definition of a single struct or data type. The trimming parser is very simple, and therefore extremely fast.

We use the snippets, and a list of structs actually used in Rekall's memory analysis plugins to isolate only those snippets which are relevant to our goal (and any of their depedencies). Therefore the total number of lines we must parse with the C parser is significantly reduced – saving a significant amount of processing time.

```
#if defined(CONFIG_TREE_RCU) || defined(CONFIG_PREEMPT_RCU)
#include <linux/rcutree.h>
#elif defined(CONFIG_TINY_RCU)
#include <linux/rcutiny.h>
#else
#error "Unknown RCU implementation specified to kernel configuration"
#endif
```

```
{
 "mro": "If:_PreASTNode:DataContainer:VisitorMixin:object"
 "conditional_blocks": [{
  "mro": "ConditionalBlock:_PreASTNode:DataContainer:VisitorMixin:object"
  "conditional_expression": "defined(CONFIG_TREE_RCU) ||
                             defined(CONFIG_PREEMPT_RCU)",
  "content": [{
   "mro": "Include:_PreASTNode:DataContainer:VisitorMixin:object",
   "absolute_path":
      "/usr/src/linux-headers-4.2.0-22-generic/include/linux/rcutree.h",
   "content": null,
   "path": "linux/rcutree.h",
   "quotes_type": "<"
   }],
  }, {
  "mro": "ConditionalBlock:_PreASTNode:DataContainer:VisitorMixin:object"
  "conditional_expression": "defined(CONFIG_TINY_RCU)",
  "content": [{
   "mro": "Include:_PreASTNode:DataContainer:VisitorMixin:object",
   "absolute_path":
      "/usr/src/linux-headers-4.2.0-22-generic/include/linux/rcutiny.h",
   "content": null,
   "path": "linux/rcutiny.h",
   "quotes_type": "<"
   }],
  }, {
  "mro": "ConditionalBlock:_PreASTNode:DataContainer:VisitorMixin:object"
  "conditional_expression": "1",
  "content": [{
   "mro": "Error:_PreASTNode:DataContainer:VisitorMixin:object"
   "message":
        "\"Unknown RCU implementation specified to kernel configuration\"",
   }],
  }],
 }
```

**Fig. 5.** An example AST produced by the preprocessing parser. In this example, different header files are included based on the value of some configuration parameters. Since the AST can not be evaluated until it is applied to the live system (when the configuration is known), the *Pre-AST* must contain both alternatives at the same time. Below is displayed the JSON serialized *Pre-AST* for the above code snippet. Each node is of a specific type (based on its python Method Resolution Order), and contains metadata and other nodes nested within it.

Additionally, this trimming phase isolates specific C code relevant for profile generation, allowing us to implement a very simplistic C parser – since it does not need to handle more complex constructs. Our C parser can only parse struct definitions, unions, enums and typedefs. It is therefore faster and more accurate than a complete C language parser written in Python.

While trimming the Pre-AST (Described in Section Trimming the Pre-AST) reduces the size of the Pre-AST, it has little impact on the time required to parse the C code. Trimming the preprocessed C file is thus used to reduce the overall time required to parse structs.

### *The C parser*

The C code within each snippet is free from macros and can be parsed into a *C-AST*. The Layout Expert's C parser must still be as authentic as possible in generating the C-AST. There are many subtle cases where parsing is particularly tricky. Some of these examples are shown in Fig. 6. In the first example we see anonymous structs defined inline, inside other struct's definitions. This example also uses unions and bit fields. Finally the first example makes use of the compiler attribute *packed* to influence field placement.

The second example shows a struct definition containing an inline array. The length of this array depends on the value of an *enum* value defined previously. The C language allows constants defined in an *enum* definition to be used in the global namespace – in this case it is used to specify a length to an array. We must parse all *enum* definitions accurately and keep a record of all their fields, since they may influence struct definitions encountered later.

arch/x86/include/asm/desc_defs.h (line 22)

```
struct desc_struct {
  union {
    struct {
      unsigned int a;
      unsigned int b;
    };
    struct {
      u16 limit0;
      u16 base0;
      unsigned base1: 8, type: 4, s: 1, dpl: 2, p: 1;
      unsigned limit: 4, avl: 1, l: 1, d: 1, g: 1, base2: 8;
    };
  };
} __attribute__((packed));
```

include/linux/sched.h (line 1337)

```
enum perf_event_task_context {
  perf_invalid_context = -1,
  perf_hw_context = 0,
  perf_sw_context,
  perf_nr_task_contexts,
};

task_struct {
  (...)
  struct perf_event_context *perf_event_ctxp[perf_nr_task_contexts];
  (...)
};
```

include/uapi/linux/timex.h (line 90)

```
struct timex {
  (...)
  int   :32; int   :32; int   :32; int   :32;
  int   :32; int   :32; int   :32; int   :32;
  int   :32; int   :32; int   :32;
};
```

**Fig. 6.** Some example of particularly tricky C code used in the Linux Kernel. In the first example we see a union of two anonymous structs. One of these structs contains bit fields. The entire struct is packed too. The second example shows a struct containing an array of pointers with a length depending the incrementing value of an *enum*. The last example shows the use of anonymous bit fields.

*Layout model*

The C compiler plans the memory layout of each newly defined struct according to a number of complex and sometimes apparently arbitrary rules. While at first it might appear that fields in structs can be laid out arbitrarily by the compiler, in fact, fields are required to be laid out in a consistent reproducible manner in order to abide by the platform's Application Binary Interface (ABI) (Matz et al., 2012). There are some cases where software must depend on the exact alignment of struct fields in memory, for example in order to pass structured data from one program to another. This often happens, even when programs communicate with the kernel. For example, the Linux *stat()* system call expects a *struct stat* pointer prepared from userspace code to be passed into kernel space through a system call. Therefore kernel and userspace code must agree on struct layout, even if the userspace code was built by a compiler other than GCC. The Linux Platform ABI contains rules about field alignments and sizes in C structs (Matz et al., 2012).

Therefore, in general, it should be possible to use the platform's ABI rules to predict the exact memory layout of structs in a consistent and reproducible manner. However, GCC also provides a number of methods for programmers to manipulate the precise alignment and padding before each struct member, via the *__attribute__* directive (Von Hagen, 2006).

The precise way in which field attributes, size and the platform ABI interact all influence the *alignment model*. The ABI documentation does not describe in details all the edge cases entailing these interactions.

We have therefore developed a set of exhaustive tests to assess all the edge cases where the alignment model is difficult to predict. We have written sample C programs containing variations of structs. The programs were then

compiled with DWARF debugging information included. Finally we refined our GCC layout model by identifying discrepancies between our predictions of member alignment and the observed alignment as obtained from DWARF symbols.

There are two main attributes determining layout. These can be specified on the entire struct, or each field:

- alignment: Specifies the alignment of the start position of the element within its containing object.
- packed: This attribute specifies that the minimum default alignment of contained objects is one byte – unless the contained object has an explicit alignment specified.

The interaction between these two attributes is quite complex. Consider the examples shown in Fig. 7. The first example illustrates the natural alignment of *int* fields to 32 bits. If the struct is not packed, GCC will align *int*s to 32 bits, and therefore create a padding past the *char* field. Similarly a *long* is naturally aligned to 64 bits, hence the third field will be padded.

In the second example we see that the struct is aligned to the smallest alignment of its members, when repeated in an array. In this case a char.

The third example illustrates the alignment of a struct containing an int, is also 8 bytes. Therefore inserting the struct into another struct will start it on an 8 byte alignment.

In the 4th example, the struct is packed. This means that the default alignment of its members is one bytes. However, the inner struct is specified to be aligned on 8 byte boundary. Therefore that field is still aligned to 8 bytes (i.e. a field's explicit alignment supersedes a packed directive).

The 5th example specifies that the inner struct should have a field, H, which is aligned to 8 bytes. However this is only specified relative to the beginning of its containing struct, which does not explicitly specify an alignment – therefore it is packed immediately after the struct. Note that the field H is not actually aligned in absolute memory at all – despite begin explicitly declared as aligned! It is only aligned relative to its (shifted and unaligned) container.

Examples 6 and 7 illustrate the difference between attributes applied to the struct itself and those applied to a variable of this type. The packed attribute causes the struct to be packed but has no effect when applied to a static variable of this type.

Structs can also declare bit fields. A bit field is a struct member which only contains a fixed number of bits. The compiler lays the bitfield inside a larger type (such as an int or char) depending on how many bits are requires. Fig. 7 also illustrates some interesting situations of bitfield alignment. In case 8, we can see that the compiler will try to fit bit fields into the previous item if they can fit (an int is 32 bits wide – so it can contain 2 fields of 30 and 2 bits each). However, in case 9, there are 3 bits required, making the compiler push the *int* bitfield into the next 32 bit slot and creating a small 2 bit pad in the first field.

Case 10 illustrates the same principle: The 64 bit long bitfield does not fit inside the previous 16 bit (short) bitfield, hence the next 64 bit slot is taken, and the long (55 bit) bitfield is pushed out to be 64 bit aligned.

The next two examples illustrate how a *packed* attribute, when applied to a field, causes the field to be placed immediately following the previous bitfield with no bit padding at all. Fields not explicitly packed, are still aligned to their natural alignment. Finally the last example shows the *packed* attribute applied to the struct causing all fields within the struct to be packed.

### *Putting it all together*

We have developed a *C-AST* visitor which traverses the AST for struct definitions. For each struct, the visitor predicts the final layout of each element based on the layout model. After processing the *C-AST*, the Layout Expert creates Rekall profiles in JSON format. Rekall profiles are divided into various sections (The Rekall Team, 2014). The layout expert stores struct definitions in the *$STRUCTS* section, and then incorporates the *System.map* and *config* file into the *$CONSTANTS* and *$CONFIG* sections respectively.

An example run is shown in Fig. 8. Resident memory use for the *Layout Expert* was around 150 Mb and total execution time was around a minute. Note that while building the *Pre-AST* we must deliberately exclude the *generated/autoconf.h* file since it contains the current config file converted into *#define* macros. If we include this file, then our macro evaluator will override the local config with the one used by the original kernel headers package during the preprocessing phase.

## Results

The *Layout Expert* finally produces a workable profile which can be provided to Rekall in order to analyze live memory (via for example the */proc/kcore* mechanism). On a system with */proc/kcore* present, there is no need to compile a kernel module for acquisition and live memory can be performed immediately, without needing a compiler.

In order to fully evaluate the accuracy of the Layout Expert's profile generation technique we have compiled a Linux kernel from source (Kernel version 4.2.0) using the stock config file shipped with Ubuntu 15.10 distribution (Hertzog and Mas, 2014). We created three packages (*linux-image*, *linux-headers* and *linux-source*) labeled as "standard". We then installed them and booted our VMWare virtual machine into the standard kernel.

We used Rekall's standard profile generation script to obtain a profile from DWARF streams. In order to do this we had to install the complete compiler tool chain on the virtual machine and the kernel header package we built previously (Needless to say, installing the required packages resulted in over 400 Mb of packages downloaded and made significant changes to the virtual machine.).

We then acquired an AFF4 image using the standard Rekall *aff4acquire* plugin (Note Rekall automatically

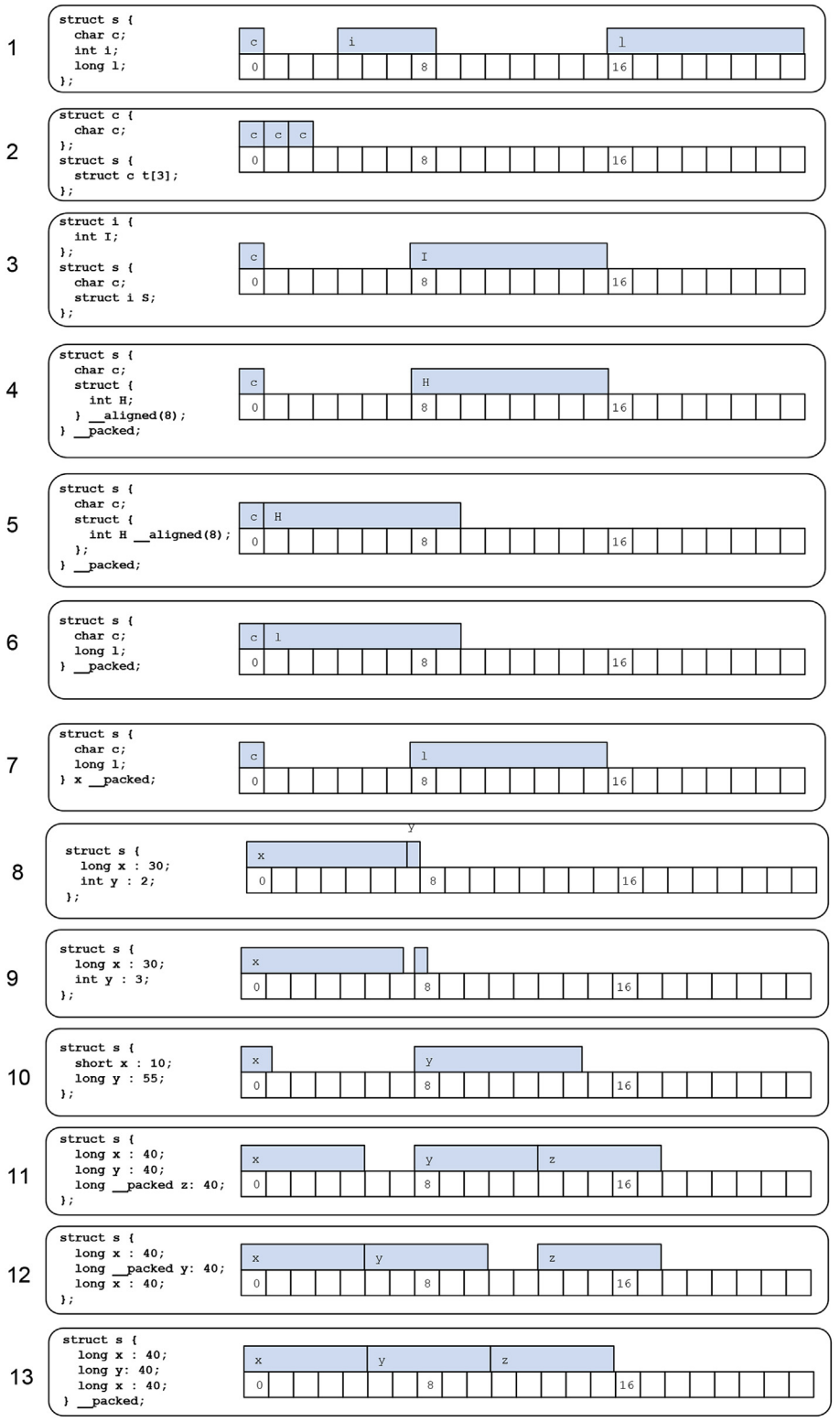



**Fig. 7.** Examples of struct layout for some combinations of field types and alignments on the AMD64 architecture.

```
$ layout_tool build_pre_ast --source_file_path ~/rekall/tools/linux/module.c \
  --linux_repository_path /usr/src/linux-headers-4.2.0-22-generic/ \
  pre_ast_4.2.0-22.json

2016-01-24 00:45:02,276 INFO     LOADING AND PARSING HEADERS
2016-01-24 00:45:02,876 INFO     Excluding include file generated/autoconf.h
2016-01-24 00:46:09,557 INFO     Completed built pre-ast forest in 67 Seconds
2016-01-24 00:46:12,893 INFO     LOADED AND PARSED
```

```
$ layout_tool make_profile --config_file_path boot/config-4.2.02.0.smp \
    --system boot/System.map-4.2.02.0.smp pre_ast_4.2.0-22.json profile.json

2016-01-23 09:44:29,416 INFO     LOADING PREPROCESSOR AST FROM: pre_ast_4.2.0...
2016-01-23 09:44:34,494 INFO     DONE
2016-01-23 09:44:34,495 INFO     LINKING INCLUDES
2016-01-23 09:44:34,937 INFO     LINKED
2016-01-23 09:44:34,937 INFO     EXTRACTING CONFIG FLAGS
2016-01-23 09:44:34,994 INFO     EXTRACTED
2016-01-23 09:44:35,108 INFO     PREPROCESSING
2016-01-23 09:44:50,856 INFO     PREPROCESSED
2016-01-23 09:44:50,856 INFO     Completed preprocessing pre-ast in 16 Seconds
2016-01-23 09:44:50,856 INFO     GENERATING PURE C FILE
2016-01-23 09:44:53,047 INFO     GENERATED
2016-01-23 09:44:53,048 INFO     Completed generating pure C file in 2 Seconds
2016-01-23 09:44:53,048 INFO     TRIMMING C FILE
2016-01-23 09:45:14,340 INFO     Completed trimming C file in 21 Seconds
2016-01-23 09:45:14,341 INFO     TRIMMED C FILE
2016-01-23 09:45:14,354 INFO     PARSING STRUCTS
2016-01-23 09:45:37,853 INFO     Completed parsing struct layouts in 23 Seconds
2016-01-23 09:45:37,853 INFO     PARSED
2016-01-23 09:45:37,853 INFO     GENERATING PROFILE
2016-01-23 09:45:37,949 INFO     Exporting 627 structs
2016-01-23 09:45:38,763 INFO     GENERATED
```

**Fig. 8.** Layout Expert output. Top: Producing the Pre-AST from kernel sources in preparation. This step only requires access to the kernel sources and Rekall's dummy module's source. Bottom: Parsing Pre-AST to generate a profile on the live system. This step combines the kernel config and system map with the *Pre-AST* file to produce a working profile.

acquires the system.map and kernel config into the AFF4 image).

Next we modified the kernel configuration by running *make menuconfig* in the kernel source tree. We wanted to change a very pervasive kernel parameter which will result in a large change for many data structures. As can be seen in Fig. 1 the *CONFIG_SMP* option is a good choice, as it is very pervasive and therefore radically changes many different structs. We repeated the build with this option disabled, rebooted and rebuilt a profile in the usual way. We also acquired a new memory image with a non-SMP kernel (i.e. single CPU support only).

Next we used the Layout Expert to build the *Pre-AST* and trim it from the kernel source tree. The resulting *trimmed Pre-AST* was compressed with *gzip* (total size 5.6 mb uncompressed, 0.57 mb compressed). Finally we used the Layout Expert to create Rekall profiles using both acquired config files (the one with SMP enabled and the one without). Resident memory size for the Layout Expert during the profile generation phase was 150 Mb as measured with the top command.

Recalling the *task_struct* struct previously depicted in Fig. 1, we examined a small snippet of the profile generated by the Layout Expert (Shown in Fig. 9). As can be seen, the field positioning varies greatly between the SMP and non-SMP cases.

As a first test of the Layout Expert's profiles we ran common Rekall plugins (*pslist*, *modules* and *lsof*) on the acquired memory images. All these plugins produced identical output when using the profile built by the Layout Expert and those built using the traditional DWARF extraction method.

Since Rekall plugins only use some fields in the struct, the ultimate test for the accuracy of the Layout Expert is to compare the complete struct layouts produced by it with the struct layouts produced by the traditional DWARF extraction method.

Fig. 10 show the same snippet from the *task_struct* struct as generated using the traditional DWARF extraction process. As we can see the offsets of all fields are identical, although the two tools use different aliases for the basic types (e.g. *unsigned long long* as compared to *long long unsigned int*).

Another difference is that the profiles extracted using DWARF symbols include many structs that are not required by Rekall. They were incidentally added into the DWARF stream during the module building process but are never used by Rekall.

```
"task_struct": [6400, {
 "acct_rss_mem1": [1688, ["unsigned long long", {}]],
 "acct_timexpd": [1704, ["unsigned long", {}]],
 "acct_vm_mem1": [1696, ["unsigned long long", {}]],
 "active_mm": [744, ["Pointer", {
     "target": "mm_struct"
  }]],
 "alloc_lock": [1536, ["spinlock", {}]],
 "atomic_flags": [824, ["unsigned long", {}]],
 "audit_context": [1496, ["Pointer", {
  "target": "audit_context"
  }]],
 "backing_dev_info": [1600, ["Pointer", {
     "target": "backing_dev_info"
  }]],
 "bio_list": [1576, ["Pointer", {
  "target": "bio_list"
  }]],
 "blocked": [1400, ["sigset_t", {}]],
 "btrace_seq": [664, ["unsigned int", {}]],
 "cg_list": [1744, ["list_head", {}]],
 "cgroups": [1736, ["Pointer", {
     "target": "css_set"
  }]],
```

```
"task_struct": [6784, {
 "acct_rss_mem1": [1896, ["unsigned long long", {}]],
 "acct_timexpd": [1912, ["unsigned long", {}]],
 "acct_vm_mem1": [1904, ["unsigned long long", {}]],
 "active_mm": [928, ["Pointer", {
     "target": "mm_struct"
  }]],
 "alloc_lock": [1736, ["spinlock", {}]],
 "atomic_flags": [1024, ["unsigned long", {}]],
 "audit_context": [1696, ["Pointer", {
     "target": "audit_context"
  }]],
 "backing_dev_info": [1808, ["Pointer", {
     "target": "backing_dev_info"
  }]],
 "bio_list": [1784, ["Pointer", {
     "target": "bio_list"
  }]],
 "blocked": [1600, ["sigset_t", {}]],
 "btrace_seq": [760, ["unsigned int", {}]],
 "cg_list": [1952, ["list_head", {}]],
 "cgroups": [1944, ["Pointer", {
     "target": "css_set"
  }]],
```

**Fig. 9.** An extract from the Rekall profiles generated by the *Layout Expert* for the non-SMP (above) and SMP (below) configurations. The profile format lists each field name, followed by its offset within the struct followed by its type. Clearly the field offsets are very different in each case, since many fields are omitted in the non-SMP case.

## Discussion

The Layout Expert is implemented in python using the *pyparsing* library (Various, 2015a, 2015b). We initially considered using the LLVM parser instead of developing our own implementation (Lattner and Adve, 2004). However, the LLVM parser does not build a preprocessing AST – rather the preprocessing step is done in tokenization time prior to parsing. Our unique approach is that the preprocessor itself is parsed into a trimmed Pre-AST. We do not require the kernel configuration to derive this Pre-AST, and therefore we can do this in advance before encountering the target system. Using the LLVM parser we would need to include the entire kernel source tree in the live memory analysis tool, since the LLVM tool requires the entire kernel configuration before any processing of source files can take place.

One of the main assumptions in our work is that we have access to the local kernel configuration and System map. If the system is maliciously attacked, the attacker may simply delete these files to make it difficult for us to perform memory analysis. We specifically do not consider this possibility in our scope. Our goal is to simply improve the normal workflow for incident response. Future work can derive the system map and configuration parameters from other aspects of the system which can be more trusted.

Another limitation in our technique is the speed of processing. We implemented the Layout Expert in Python

```
"task_struct": [6784, {
 "acct_rss_mem1": [1896, ["long long unsigned int"]],
 "acct_timexpd": [1912, ["long unsigned int"]],
 "acct_vm_mem1": [1904, ["long long unsigned int"]],
 "active_mm": [928, ["Pointer", {
  "target": "mm_struct",
  }]],
 "alloc_lock": [1736, ["spinlock"]],
 "atomic_flags": [1024, ["long unsigned int"]],
 "audit_context": [1696, ["Pointer", {
  "target": "audit_context",
  }]],
 "backing_dev_info": [1808, ["Pointer", {
  "target": "backing_dev_info",
  }]],
 "bio_list": [1784, ["Pointer", {
  "target": "bio_list",
  }]],
 "blocked": [1600, ["sigset_t"]],
 "btrace_seq": [760, ["unsigned int"]],
 "cg_list": [1952, ["list_head"]],
```


**Fig. 10.** A Rekall profile built using the traditional DWARF extraction method for the SMP enabled kernel. As compared with Fig. 9 all the fields are identical apart from use of equivalent type definitions for integers.

due to the easy access to powerful parsing libraries. Python is, however, much slower than the GCC compiler tool chain. Therefore, we had to implement certain strategies to optimize performance. For example, trimming the preprocessed C file can be seen as merely an optimization step, since it reduces the total size of code we need to parse. Despite these difficulties we consider the total processing time of around 1 min to be acceptable in practice.

The Layout Expert still needs to create a single Pre-AST for every potential version of the kernel source tree. Our solution accounts for variations in kernel configuration, but not kernel versions. A complete solution to live memory analysis would require an extensive library of released versions of the Linux tree. However in contrast to current approachs that require a distinct kernel profile for each configuration variation at a single kernel version, our library would only require a single Pre-AST file for each version. This file already supports all possible configurations.

We assume that users started off from a released version, obtained from a distribution and then they only modified their configuration. Therefore there should be a relatively small number of possible kernel releases to consider (e.g. Ubuntu release tag commits in the git repository). In this work we do not deal with the possibility that a user may have checked their source code at an arbitrary git commit, or applied custom patches to their source tree (For example, users typically apply kernel patches in order to support different hardware). In practice however, many patches rarely change core kernel components such as *task_struct* and so using a *Pre-AST* from the regular unpatched kernel might still work for typical memory analysis needs.

### *Further applications*

The ability to accurately predict the layout of kernel structures is not only critical for analysis – it is also critical for the initial acquisition phase itself.

On Linux there are two main modes of memory acquisition. The first relies on the presence of the */proc/kcore* device, which exposes physical memory to userland. However, in more secure installations, this device is disabled, requiring a special kernel module to be loaded in order to expose and acquire physical memory.

In order to load a kernel module into a running kernel, the module must be compiled with the precise kernel configuration that was used to build the kernel. The main reason is that the kernel module must interact with the running kernel's structs and so must have the same struct layout as the running kernel. Stüttgen and Cohen (2014) have demonstrated an interesting technique where code is injected into an existing kernel module (a parasite) which would already have the correct kernel struct layout to enable the module to be safely inserted. However their technique depends on the availability of a suitable parasite module on the running system.

The layout expert, however, can solve this problem entirely, since we can arrive at the correct struct layout at runtime. We can therefore modify an existing linux kernel module to contain the correct struct layout on the running system before inserting the kernel module. This makes the layout expert useful in acquisition situations as well, and is an exciting future application of this technology.

## Conclusions

Traditionally, memory analysis on the Linux platform has been cumbersome due to the need for advanced preparations: Profiles and drivers were required to be compiled in advance before they can be run on the system. This practical limitation made incident response on Linux platforms difficult.

The main reason for this difficulty is the inability to predict in advance the memory layout of kernel structures, without the assistance of the compiler itself and it's emitted debug information.

For the first time, we offer a self contained solution which can work without relying on the compiler toolchain, without installing any additional tools on the target system and without contacting external services. The *Layout Expert* project is able to calculate the memory layout of critical kernel parameters from the kernel configuration found on the actual target machine itself. The tool is able to automatically create fully functional profiles for live analysis on Linux, without requiring any advanced preparations.

We have released our tool under an open source licence and have contributed it to the Rekall memory forensic suite.

## References


Case A, Marziale L, Richard III GG. Dynamic recreation of kernel data structures for live forensics. Digit Investig 2010;7:S32–40.

Cohen M. Installing rekall on windows. 2015. http://rekall-forensic.blogspot.ch/2015/09/installing-rekall-on-windows.html.

Cohen M, Bilby D, Caronni G. Distributed forensics and incident response in the enterprise. Digit Investig 2011;8:S101–10.

Cohen MI. Characterization of the windows kernel version variability for accurate memory analysis. Digit Investig 2015b;12:S38–49.

DWARF debugging information format committee. DWARF debugging information format. version 4. Free Standards Group; 2010. http://dwarfstd.org/.

Hertzog R, Mas R. The Debian administrator's handbook, Debian Wheezy from discovery to mastery. 2014. Lulu.com.

Kernighan BW, Ritchie DM, Ejeklint P. The C programming language, vol. 2. prentice-Hall Englewood Cliffs; 1988.

Lattner C, Adve V. Llvm: A compilation framework for lifelong program analysis & transformation. In: Code generation and optimization, 2004. CGO 2004. International symposium on. IEEE; 2004. p. 75–86.

Ligh MH, Case A, Levy J, Walters A. The art of memory forensics: detecting malware and threats in windows, linux, and mac memory. 1st ed. Wiley Publishing; 2014.

Lin Z, Rhee J, Zhang X, Xu D, Jiang X. Siggraph: brute force scanning of kernel data structure instances using graph-based signatures. In: NDSS Symposium; 2011.

Matz M, Hubicka J, Jaeger A, Mitchell M. System V application binary interface. AMD64 architecture processor supplement. 2012.

Moser A, Cohen MI. Hunting in the enterprise: forensic triage and incident response. Digit Investig 2013;10(2):89–98.

Okolica J, Peterson GL. Windows operating systems agnostic memory analysis. Digit Investig 2010;7:S48–56.

Schuster, Andreas. PTFinder version 0.3.05. 2007. http://computer.forensikblog.de/en/2007/11/ptfinder-version-0305.html.

Secondlook. Second look: advanced linux threat detection. 2015. http://secondlookforensics.com/.

Stüttgen J, Cohen M. Robust linux memory acquisition with minimal target impact. Digit Investig 2014;11:S112–9.

The Rekall Team. The Rekall memory forensic framework. 2014. http://www.rekall-forensic.com/.

The Volatility Foundation. The volatility framework. 2014. http://www.volatilityfoundation.org/.

Various. GRR rapid response: remote live forensics for incident response. 2015. https://github.com/google/grr.

Various. Pyparsing Wiki home. 2015. https://pyparsing.wikispaces.com/.

Von Hagen W. The definitive guide to GCC. Apress; 2006.