Analyzing changes to the binary interface exposed by the Kernel to its modules

Kernel Recipes 2019

Jessica Yu -- Dodji Seketeli -- Matthias Männich

What makes up the kernel ABI?

- Low level binary interface between the kernel and its modules
- Set of exported symbols (functions, variables) and their symbol versions (CRCs)
- Data structure layouts, offsets, size, alignment
- ABI tools check these structural expectations

Kernel ABI - why care?

- In a perfect world, all modules would be in-tree and upstream...
- Unfortunately, reality is more complicated

Kernel ABI - why care?

- Distributors care about maintaining kABI stability for out-of-tree modules from partners/vendors
- Prevent needed third party modules from breaking with routine kernel updates

Kernel ABI - why care?

- Decouple development of the kernel and its modules
- Provide a single kernel ABI / API for ecosystem of vendor modules

Upstream tools for kernel ABI checks

- modversions/genksyms (CONFIG_MODVERSIONS)

Modversions: Limitations

- yacc/lex based parser (genksyms)
- Prone to report false positive kABI breakages
- Maintenance pain
- Limited reporting of kABI breakage

Modversions: Limitations

```
KABI: symbol bio_trim(vmlinux) changed crc from 0xea9cb7e9 to 0x595bb017

block/bio.c:1891: warning: bio_trim: modversion changed because of changes in
struct blk_mq_tags
```

Modversions: Limitations

```
diff --git a/include/linux/livepatch.h b/include/linux/livepatch.h
index 273400814020..326fb541588a 100644
--- a/include/linux/livepatch.h
+++ b/include/linux/livepatch.h
@@ -153,8 +153,7 @@ struct klp_patch {
     struct list_head list;
     struct kobject kobj;
     struct list_head obj_list;
-    bool enabled;
-    bool forced;
+    bool enabled, forced;
     struct work_struct free_work;
     struct completion finish;
 };
```

Modversions: Limitations

```
linux/kernel/livepatch/core.c:1022: error: klp_enable_patch: modversion
changed because of changes in struct klp_patch

Export klp_enable_patch == <int klp_enable_patch ( struct klp_patch {
... struct list_head obj_list ; bool enabled , forced ; struct work_struct
free_work ; ... >

Export klp_enable_patch == <int klp_enable_patch ( struct klp_patch {
... struct list_head obj_list ;  bool enabled ; bool forced ;
 struct work_struct free_work ; ... >
```

- As a result, CRC also changes, although the binary interface has stayed the same

Modversions: Limitations

```
 <2><9916d7>: Abbrev Number: 14 (DW_TAG_member)
    <9916d8>   DW_AT_name        : (indirect string, offset: 0x212486):
enabled
    <9916dc>   DW_AT_decl_file   : 203
    <9916dd>   DW_AT_decl_line   : 156
    <9916de>   DW_AT_type        : <0x983849>
    <9916e2>   DW_AT_data_member_location: 120
 <2><9916e3>: Abbrev Number: 14 (DW_TAG_member)
    <9916e4>   DW_AT_name        : (indirect string, offset: 0x64b0f): forced
    <9916e8>   DW_AT_decl_file   : 203
    <9916e9>   DW_AT_decl_line   : 156
    <9916ea>   DW_AT_type        : <0x983849>
    <9916ee>   DW_AT_data_member_location: 121
```

Modversions: Limitations

- Similar things happen when a struct suddenly becomes defined (inclusion of header file)

  ```
  Export for_each_kernel_tracepoint == <void for_each_kernel_tracepoint ( void
  ( * ) ( struct tracepoint { … struct static_key_mod { UNKNOWN } * next ;
  …>
  ```

  Move the definition of static_key_mod to a header file included by kernel/tracepoint.c, you get:

  ```
  Export for_each_kernel_tracepoint == <void for_each_kernel_tracepoint ( void
  ( * ) ( struct tracepoint { … struct static_key_mod { struct static_key_mod
  * next ; struct jump_entry * entries ; struct module { enum module_state {
  MODULE_STATE_LIVE , …>
  ```

- As a result, CRC also changes, although the binary interface has stayed the same

Modversions: Limitations

```
 $ echo 'struct foo { int bar; };' | ./scripts/genksyms/genksyms -d
Defn for struct foo == <struct foo { int bar ; } >
Hash table occupancy 1/4096 = 0.000244141

$ echo 'struct __attribute__((packed)) foo { int bar; };' |
./scripts/genksyms/genksyms -d
Hash table occupancy 0/4096 = 0
```

Source: https://lore.kernel.org/lkml/CAKwvOdnJAApaUhTQs7w_VjSeYBQa0c-TNxRB4xPLi0Y0sOQMMQ@mail.gmail.com/

Distro requirements

- ABI tracking for a (sub)set of exported symbols
- Normally don’t care about ABI of in-tree only or inter-driver/inter-module symbols (symbol whitelist)
- Human-readable kABI reports, easily pinpoint source of kABI breakages and save developer time
- Runtime ABI checks (checking symbol CRCs at module load time)
- Doesn’t extend kernel/package build time by too much
- ...

Enters Libabigail

- “ABI Generic Analysis and Instrumentation Library”
  - Framework to analyze ABI by looking at binaries directly
- It’s basically a library
  - Reads ELF and DWARF information
  - Builds an internal in-memory representation of ABI artifacts (a.k.a ABI/IR)
    - Functions, variables, types, ELF symbols
- Can compare two ABI IRs
  - The result of the comparison is also an IR (a.k.a DIFF/IR)
- The aim is to analyze the graphs represented by the IRs to emit useful information
- Different tools use the library in specific ways for specific purposes
  - Initially tailored to support analysis of userspace shared libraries
- Jessica and Matthias recently joined to help make the framework grok Linux kernel binaries

Libabigail Kernel Support

Or what's specific about it ...

- Kernel binaries have several symbol tables
  - .symtab, __ksymtab, __ksymtab_gpl
  - that's where we find symbols of functions/variables that matter
  - possibly various formats
    - since v4.19 new format possible on AARCH64

- Kernel made of vmlinux + thousands of modules and hundreds of thousands of types
  - We want to build one in-memory representation for vmlinux + modules
  - We want to be able to load two Kernel IR in memory at one point in time

- Somewhat bigger than your average shared library out there …
  - Challenging to be fast enough but we are regularly making progress

Libabigail

Typical Breakages -- Added Function

```
 struct my_type {
   char str[1024];
 } my_type;

 struct my_struct {
   int a;
   struct my_type b;
 } my_struct;

 enum my_enum {
   A, C, B
 };

 void func_str(struct my_struct   s) {}
 void func_ptr(struct my_struct * s) {}
 void func_enu(enum   my_enum     e) {}

+void new_func() {}
```

```
Functions changes summary: 0 Removed, 0 Changed, 1 Added function
Variables changes summary: 0 Removed, 0 Changed, 0 Added variable

1 Added function:

  'function void new_func()'    {new_func}
```

ABI Breakage	yes, but ABI still compatible
Mitigation	n/a
Notes	`new_func()` becomes part of the ABI. An update of the ABI representation is required to catch future breakages of `new_func()`.

Libabigail

Typical Breakages -- Sort enum values

```
struct my_type {
  char str[1024];
} my_type;

struct my_struct {
  int a;
  struct my_type b;
} my_struct;

enum my_enum {
-  A, C, B
+  A, B, C
};

void func_str(struct my_struct   s) {}
void func_ptr(struct my_struct * s) {}
void func_enu(enum   my_enum     e) {}
```

```
Functions changes summary: 0 Removed, 1 Changed, 0 Added function
Variables changes summary: 0 Removed, 0 Changed, 0 Added variable

1 function with some indirect sub-type change:

  [C]'function void func_enu(my_enum)' at test.c:16:1 has indirect sub-type
changes:
    parameter 1 of type 'enum my_enum' has sub-type changes:
      type size hasn't changed
      2 enumerator changes:
        'my_enum::C' from value '1' to '2' at test.c:10:1
        'my_enum::B' from value '2' to '1' at test.c:10:1
```

ABI Breakage	no, but API breakage
Mitigation	Just don't.
Notes	Again, this is not a strict ABI breakage, but still discovered as breakage. In this case, the change is not considered harmless.

Libabigail

Typical Breakages -- Removed Function

```
struct my_type {
  char str[1024];
} my_type;

struct my_struct {
  int a;
  struct my_type b;
} my_struct;

enum my_enum {
  A, C, B
};

void func_str(struct my_struct   s) {}
void func_ptr(struct my_struct * s) {}
-void func_enu(enum   my_enum     e) {}
```

```
Functions changes summary: 1 Removed, 0 Changed, 0 Added function
Variables changes summary: 0 Removed, 0 Changed, 0 Added variable

1 Removed function:

  'function void func_enu(my_enum)'    {func_enu}
```

ABI Breakage	yes
Mitigation	Add back the symbol (maybe forward if it was a rename).
Notes	

Libabigail

Typical Breakages -- Add struct member

```
 struct my_type {
   char str[1024];
 } my_type;

 struct my_struct {
   int a;
   struct my_type b;
+  int new_member;
 } my_struct;

 enum my_enum {
   A, C, B
 };

 void func_str(struct my_struct   s) {}
 void func_ptr(struct my_struct * s) {}
 void func_enu(enum   my_enum     e) {}
```

```
Functions changes summary: 0 Removed, 2 Changed, 0 Added functions
Variables changes summary: 0 Removed, 0 Changed, 0 Added variable

2 functions with some indirect sub-type change:

  [C]'function void func_str(my_struct)' at test.c:15:1 has indirect sub-type changes:
    parameter 1 of type 'struct my_struct' has sub-type changes:
      type size changed from 8224 to 8256 (in bits)
      1 data member insertion:
        'int my_struct::new_member', at offset 8224 (in bits) at test.c:8:

  [C]'function void func_ptr(my_struct*)' at test.c:16:1 has indirect sub-type changes:
    parameter 1 of type 'my_struct*' has sub-type changes:
      in pointed to type 'struct my_struct' at test.c:5:1:
      [...]
```

ABI Breakage	yes, but strictly only for `func_str()`
Mitigation	Add padding to data structures potentially being affected.
Notes	`func_ptr()` does not actually suffer from an ABI breakage at that very definition. It might be affected when the parameter gets dereferenced or if its size is taken. Libabigail diagnoses this because it can. It would not if the full definition of `my_struct` would not be available.

Libabigail

Typical Breakages -- Struct member name change

```
 struct my_type {
   char str[1024];
 } my_type;

 struct my_struct {
   int a;
-  struct my_type b;
+  struct my_type c;
 } my_struct;

 enum my_enum {
   A, C, B
 };

 void func_str(struct my_struct   s) {}
 void func_ptr(struct my_struct * s) {}
 void func_enu(enum   my_enum     e) {}
```

```
Functions changes summary: 0 Removed, 0 Changed (2 filtered out),
                           0 Added functions
Variables changes summary: 0 Removed, 0 Changed, 0 Added variable
```

ABI Breakage	no, but API breakage
Mitigation	From an ABI point of view this is sane.
Notes	The rename is detected, but **considered harmless**. Most likely the rename is not essential for the intent of the actual patch. So, reverting the rename is probably a good idea.

Libabigail

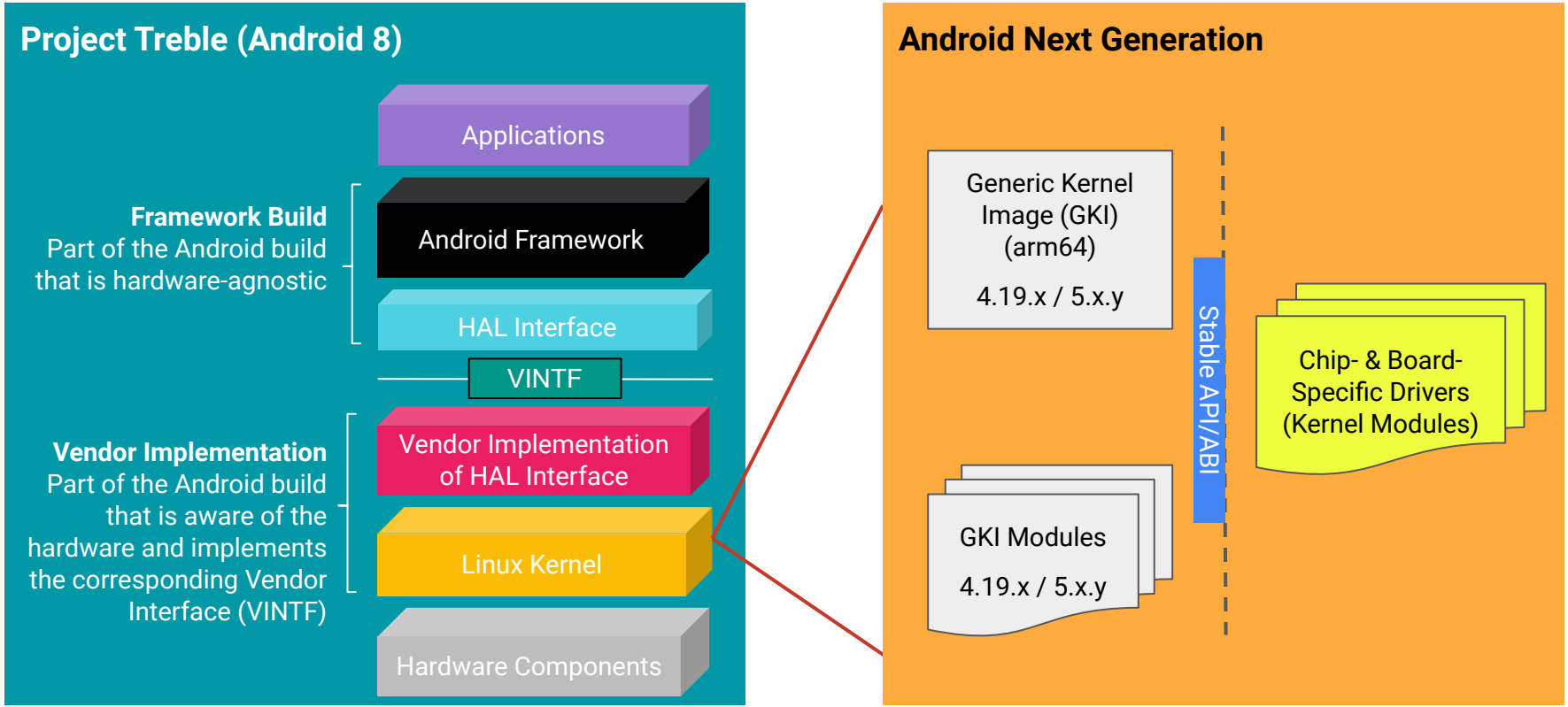
Typical Breakages -- Type change (with identical memory layout)

```
 struct my_type {
   char str[1024];
 } my_type;

 struct my_struct {
   int a;
-  struct my_type b;
+  char b[1024];
 } my_struct;

 enum my_enum {
   A, C, B
 };

 void func_str(struct my_struct   s) {}
 void func_ptr(struct my_struct * s) {}
 void func_enu(enum   my_enum     e) {}
```

```
Functions changes summary: 0 Removed, 2 Changed, 0 Added functions
Variables changes summary: 0 Removed, 0 Changed, 0 Added variable

2 functions with some indirect sub-type change:

  [C]'function void func_ptr(my_struct*)' at test.c:15:1 has indirect sub-type changes:
    parameter 1 of type 'my_struct*' has sub-type changes:
      in pointed to type 'struct my_struct' at test.c:5:1:
        type size hasn't changed
        1 data member change:
         type of 'my_type my_struct::b' changed:
           entity changed from 'struct my_type' to 'char[1024]'
[...]
```

ABI Breakage	no, but API breakage
Mitigation	Just don't.
Notes	Again, this is not a strict ABI breakage, but still discovered as breakage. In this case, the change is not considered harmless.

Stable ABIs for Android Kernels



Stable ABI within Boundaries

and how Android implements that*

Branches	Configuration	Toolchain	Scope
○ Only keep ABI stable within major upstream branch ○ E.g. LTS 4.9, 4.14, 4.19, 5.x	○ Single Kernel Configuration ○ Suitable for all vendors ○ Configuration changes allowed if they don't break ABI	○ Single Toolchain ○ Hermetic Build	○ Define what is part of the ABI ○ Whitelist ○ Suppression
○ android-4.19 ○ android-5.x *	○ Generic Kernel Image (GKI) configuration	○ Clang Build (only) ○ Hermetic Toolchain* (enforced by build wrapper)	○ Observable ABI ○ Whitelists * ○ **Symbol Namespaces** *

* some of these are still work in progress and to be finalized / implemented

Integration into the Android Kernel Build

<table>
<tr><th colspan="3">build_abi.sh</th></tr>
<tr><th colspan="2">build.sh</th><th>ABI Tooling</th></tr>
<tr>
<td>○ Setup (hermetic) build environment
○ Toolchain
○ Cross Compiler
○ Build Dir / Dist Dir</td>
<td>○ make mrproper
○ make <defconfig>
○ make

○ create Android Kernel distribution</td>
<td>○ abidw --linux-tree out/

○ abidiff abi.xml abi-base.xml

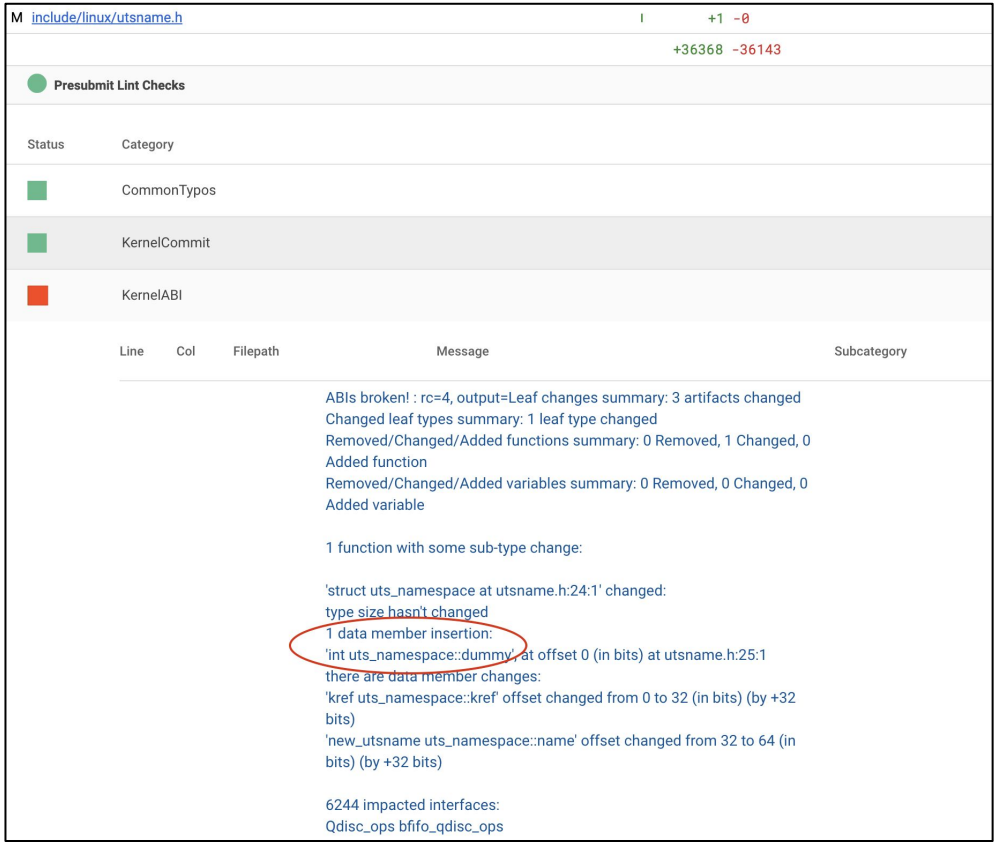
○ create abi report</td>
</tr>
</table>

```
$ repo init -u <url> -b <branch>     # initialize workspace

$ repo sync                          # get sources, toolchain, dependencies, etc.

$ build/build_abi.sh                 # build and validate ABI
```

Monitoring and Enforcement

- Define a baseline ABI
- Keep it along with your sources
- Establish ABI checking (e.g. `build_abi.sh`) as mandatory test before merging
- Changes targeting Android Common Kernels have to pass this test in AOSP Gerrit

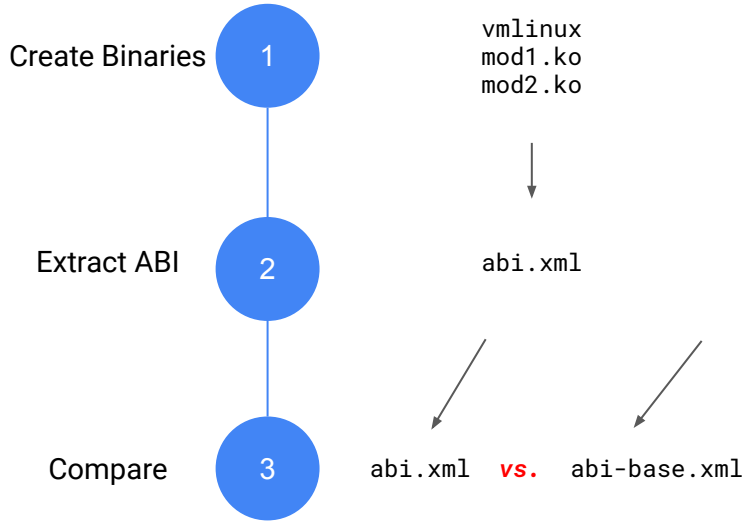
```
--- a/include/linux/utsname.h
+++ b/include/linux/utsname.h
@@ -22,6 +22,7 @@ struct user_namespace;
 extern struct user_namespace init_user_ns;

 struct uts_namespace {
+       int dummy;
        struct kref kref;
        struct new_utsname name;
        struct user_namespace *user_ns;
```



Libabigail

"Application Binary Interface Generic Analysis and Instrumentation Library"
https://sourceware.org/libabigail/

- Library and set of tools to analyze ABIs of binaries
- Allows serializing, deserializing and comparing of ABI representation
- Considers ELF symbols along with DWARF information
- Linux Kernel Support is fairly new, but works pretty good (considers ***ksymtab*** instead of ELF symbol table)
- Support for 4.19+ Kernels is almost completed



Libabigail

ABI Representation

```
struct my_type {
  char str[1024];
} my_type;

struct my_struct {
  int a;
  struct my_type b;
} my_struct;

enum my_enum {
  A, C, B
};

void func_str(struct my_struct   s) {}
void func_ptr(struct my_struct * s) {}
void func_enu(enum   my_enum     e) {}
```

```
<abi-corpus path='test.o' architecture='elf-amd-x86_64'>

  <elf-function-symbols>
    <elf-symbol name='func_enu' type='func-type'/>
    <elf-symbol name='func_ptr' type='func-type'/>
    <elf-symbol name='func_str' type='func-type'/>
  </elf-function-symbols>
  <elf-variable-symbols>
    <elf-symbol name='my_struct' size='1028' type='object-type'/>
    <elf-symbol name='my_type' size='1024' type='object-type'/>
  </elf-variable-symbols>

[...]
```

Libabigail

ABI Representation

```
struct my_type {
  char str[1024];
} my_type;

struct my_struct {
  int a;
  struct my_type b;
} my_struct;

enum my_enum {
  A, C, B
};

void func_str(struct my_struct   s) {}
void func_ptr(struct my_struct * s) {}
void func_enu(enum   my_enum     e) {}
```

```
[...]

  <abi-instr version='1.0' address-size='64' language='LANG_C99'>

    <type-decl name='void' id='type-id-1'/>

    <type-decl name='enum-int' size-in-bits='32' id='type-id-2'/>

    <enum-decl name='my_enum' line='10' column='1' id='type-id-3'>
      <underlying-type type-id='type-id-2'/>
      <enumerator name='A' value='0'/>
      <enumerator name='C' value='1'/>
      <enumerator name='B' value='2'/>
    </enum-decl>

    <function-decl name='func_enu' line='16' column='1'
                   size-in-bits='64' elf-symbol-id='func_enu'>
      <parameter type-id='type-id-3' name='e' line='16' column='1'/>
      <return type-id='type-id-1'/>  <!-- void ->
    </function-decl>

[...]
```

Libabigail

ABI Representation

```
struct my_type {
  char str[1024];
} my_type;

struct my_struct {
  int a;
  struct my_type b;
} my_struct;

enum my_enum {
  A, C, B
};

void func_str(struct my_struct   s) {}
void func_ptr(struct my_struct * s) {}
void func_enu(enum   my_enum     e) {}
```

```
[...]
    <class-decl name='my_struct' size-in-bits='8224' is-struct='yes'
                line='5' column='1' id='type-id-4'>
      <data-member layout-offset-in-bits='0'>
        <var-decl name='a' type-id='type-id-5' line='6' column='1'/>
      </data-member>
      <data-member layout-offset-in-bits='32'>
        <var-decl name='b' type-id='type-id-6' line='7' column='1'/>
      </data-member>
    </class-decl>
    <type-decl name='int' size-in-bits='32' id='type-id-5'/>
    <class-decl name='my_type' size-in-bits='8192' [...]</class-decl>


    <pointer-type-def type-id='type-id-4' size-in-bits='64' id='type-id-11'/>
    <function-decl name='func_ptr' line='15' column='1'
                   size-in-bits='64' elf-symbol-id='func_ptr'>
      <parameter type-id='type-id-11' name='s' line='15' column='1'/>
      <return type-id='type-id-1'/>
    </function-decl>

[...]
```

Unhandled Cases

Untagged enums

include/linux/mm.h

```
enum {
    REGION_INTERSECTS,
    REGION_DISJOINT,
    REGION_MIXED,
};

/* returns one of the above values */
int region_intersects(resource_size_t offset,
                      size_t size,
                      unsigned long flags,
                      unsigned long desc);
```

How to handle sorting of such an enum now?

```
enum {
-   REGION_INTERSECTS,
    REGION_DISJOINT,
+   REGION_INTERSECTS,
    REGION_MIXED,
};
```

Generate ABI capturing data structures (code generation or compiler plugin)

```
enum abi_enum {
    abi_REGION_INTERSECTS = REGION_INTERSECTS,
    abi_REGION_DISJOINT = REGION_DISJOINT,
    abi_REGION_MIXED = REGION_MIXED,
};

void abi_func(enum abi_enum e) { }
```

Captured as ELF symbol

Unhandled Cases

Defines

include/linux/sched.h

```
/* Used in tsk->state: */
#define TASK_RUNNING            0x0000
#define TASK_INTERRUPTIBLE      0x0001
#define TASK_UNINTERRUPTIBLE    0x0002
[...]
#define TASK_WAKEKILL           0x0100
[...]

/* Convenience macros for the sake of
set_current_state: */
#define TASK_KILLABLE                           \
   (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
```

How to handle changes to #defined values?

```
 #define TASK_RUNNING           0x0000
 #define TASK_INTERRUPTIBLE     0x0001
-#define TASK_UNINTERRUPTIBLE   0x0002
+#define TASK_UNINTERRUPTIBLE   0x0004

 [...]
#define TASK_KILLABLE  (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
```

Deduct defines and create trackable data structures.

```
enum abi_enum {
    abi_TASK_RUNNING = 0x0000,
    abi_TASK_INTERRUPTIBLE = 0x0001,
    abi_TASK_UNINTERRUPTIBLE = 0x0002,
    abi_TASK_WAKEKILL = 0x0100,
    abi_TASK_KILLABLE = 0x0102
};

void abi_func(enum abi_enum e) { }
```

Captured as ELF symbol

Questions?

Dodji Seketeli	dodji@seketeli.org
Jessica Yu	jeyu@kernel.org
Matthias Männich	maennich@android.com