

# Design and specification for Shiva: A *Programmable linking engine for ELF microcode patching*

```
ADR_PREL_P6_HI21: /* ((PAGE(S-A) - PAGE(P)) >> 12)
...
/* the relocation symbol reference a section header
... usually references '.text' as it's symbol.

foreach(smap current, &linker->tailq.section_maplist)
if (strcmp(smap->name, rel.symname) != 0)
continue;
shiva debug("Applying R_AARCH64_ADR_PREL_P6_HI21
rel.symname);
symval = smap->vaddr;
rel_unit = &linker->text_mem[smap.offset + rel.of
rel_addr = linker->text_vaddr + smap.offset + rel
rel_val = ELF_PAGESTART(symval + rel.addend) - EL
rel_val = rel_val >> 12;
memcpy(&insn_bytes, &rel_unit[0], sizeof(uint32_t
/*
* Re-encode the instruction with the new IMM field
*/
insn_bytes = (insn_bytes & ~(RELOC_MASK(2) << 29)
| ((rel_val & RELOC_MASK(2)) << 29) | ((rel_val
*(uint32_t *)&rel_unit[0] = insn_bytes;
return true;
```

- 1.0 – Description
- 2.0 – Shiva linker chaining
  - 2.1 Static ELF executables
- 3.0 – Shiva ELF ulexec loader
- 4.0 – Shiva runtime linking and loading of patches
  - 4.1. Proces address space requirements
  - 4.2. ET\_REL modules for patching
    - 4.2.A. DT\_SHIVA\_NEEDED entry in PT\_DYNAMIC
    - 4.2.B shiva-ld (The pre-linker)
  - 4.4. Symbol interposition on code/data
  - 4.5. Cross ELF relocation between chained linkers
  - 4.6. Declaring external variables within patch source
  - 4.7. Byte-code patching with Program Transformation directives
  - 4.8. Function splicing with Transformations
- 5.0 – Shiva external re-linking on target executablePTD
- References

## 1.0 Description

Shiva is a programmable runtime-linker that is designed to run in a *Linker Chain* alongside the regular RTLD “*/lib64/ld-linux.so*”. It is a custom program interpreter that loads ELF patches in the form of ET\_REL object files. Therefore and similarly to the dynamic linker, the patches are modular and contain the rich symbol and relocation meta-data that is necessary to transform the executable runtime environment.

The modular approach is dynamic, extensible, and elegantly follows conventions, rules, and linking policies of the existing ELF ABI toolchain for Linux.

Patches can be written purely in C when patching a symbol directly (i.e. symbol interposition). In more nuanced patching scenarios the patch may require the use of specialized C macros that instruct the

linker on where to emit instruction bytes, splice in code, and extend the text or data segment. Transformations re-landscape the program at runtime with *Shiva's transformation engine*; which is why we call it a “Programmable linking engine”.

Our goal is to be able to patch vulnerabilities within the NASA cFS (*Control flight software*). The software we are patching is a dynamically linked AArch64 ELF PIE binary. Currently our approach for microcode patching supports any AARCH64 PIE ELF binary, both dynamically linked and statically linked.

## 2.0 – Shiva Linker Chaining

Upon execution of a dynamically linked ELF executable, the kernel (See *binfmt\_elf.c*) parses the ELF program header segments, and retrieves the “Interpreter Path” from the PT\_INTERP segment.

**Image 2.1: ELF executable with PT\_INTERP: “ld-linux-aarch64.so”**

```
Elf file type is DYN (Shared object file)
Entry point 0x7e0
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz             MemSiz             Flags   Align
PHDR             0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001c0 0x00000000000001c0  R       0x8
INTERP           0x0000000000000200 0x0000000000000200 0x0000000000000200
                 0x000000000000001b 0x000000000000001b  R       0x1
  [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
LOAD             0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000a98 0x0000000000000a98  R E     0x10000
LOAD             0x0000000000000d40 0x00000000000010d40 0x00000000000010d40
                 0x00000000000002d0 0x000000000000042e0  RW      0x10000
DYNAMIC          0x0000000000000d50 0x00000000000010d50 0x00000000000010d50
                 0x0000000000000200 0x0000000000000200  RW       0x8
NOTE            0x000000000000021c 0x000000000000021c 0x000000000000021c
                 0x0000000000000044 0x0000000000000044  R        0x4
GNU_STACK        0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW       0x10
GNU_RELRO        0x0000000000000d40 0x00000000000010d40 0x00000000000010d40
                 0x00000000000002c0 0x00000000000002c0  R        0x1
```

The Dynamic linker is loaded by the kernel, and executed before anything else. The dynamic linker is responsible for loading and linking *shared object modules* (i.e. *libc.so*). It is necessary for the linking and loading of dependencies at runtime. Our approach to microcode patching introduces a secondary Dynamic linker “/lib/shiva”. The standard convention is that there is only one PT\_INTERP segment and it provides the path to the one and only *Program Interpreter*. So how do we get the kernel to load two separate dynamic linkers? The answer is **Linker Chaining**.

**Linker Chaining** is a technique that I first published in the paper titled “*Preloading the linker for fun and profit*” [1] The technique essentially explains how the *Program Interpreter* specified in the PT\_INTERP segment can be loaded and executed by the kernel, and afterwards it is responsible for loading a secondary interpreter (Probably ld-linux.so). The next Interpreter in the chain, hence the term *Linker Chaining*.

Shiva replaces the standard program interpreter ld-linux.so in PT\_INTERP and after Shiva has finished linking modular patches into the address space it is responsible for loading/mapping the original interpreter *ld-linux.so* into memory just like the kernel *binfmt\_elf.c* would do it. The Shiva runtime linker eventually passes control over to the next chained interpreter, which happens to be *ld-linux.so*. The *ld-linux.so* interpreter finishes it’s loading and linking of shared modules, and eventually passes control to the target executable.

**Image 2.2: ELF executable with PT\_INTERP: “/lib/shiva”**

```
Elf file type is DYN (Shared object file)
Entry point 0x7e0
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz             MemSiz             Flags   Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001c0 0x00000000000001c0 R       0x8
  INTERP         0x0000000000000200 0x0000000000000200 0x0000000000000200
                 0x000000000000000b 0x000000000000001b R       0x1
    [Requesting program interpreter: /lib/shiva]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000000a98 0x00000000000000a98 R E     0x10000
  LOAD           0x0000000000000d40 0x00000000000010d40 0x00000000000010d40
                 0x00000000000002d0 0x000000000000042e0 RW      0x10000
  DYNAMIC        0x0000000000000d50 0x00000000000010d50 0x00000000000010d50
                 0x0000000000000200 0x0000000000000200 RW      0x8
  NOTE           0x000000000000021c 0x000000000000021c 0x000000000000021c
                 0x0000000000000044 0x0000000000000044 R       0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000 RW      0x10
  GNU_RELRO      0x0000000000000d40 0x00000000000010d40 0x00000000000010d40
                 0x00000000000002c0 0x00000000000002c0 R       0x1
```

## 2.1 – Static ELF executables

ELF executables which have been statically linked have no need for a dynamic linker and therefore do not have a PT\_INTERP segment. In this case the Shiva Prelinker “/bin/shiva-ld” will insert a PT\_INTERP segment into the static ELF executable. This can be accomplished by converting the superfluous PT\_NOTE segment into a PT\_INTERP segment containing the path to “/lib/shiva”.

When the static executable is run the kernel will load “/lib/shiva” as the program interpreter and pass control to it. After the linking and loading of the needed micropatches Shiva will pass control to the entry point of the static executable (Instead of the entry point of “ld-linux.so”).

### 3.0 – The Shiva ELF ulexec loader

Shiva can be executed indirectly as an interpreter and directly as a regular executable.

When Shiva is executed indirectly (i.e. As another programs *program interpreter*) “/lib/shiva” is mapped into the process by the kernel as the *interpreter program* of the program that is being executed.

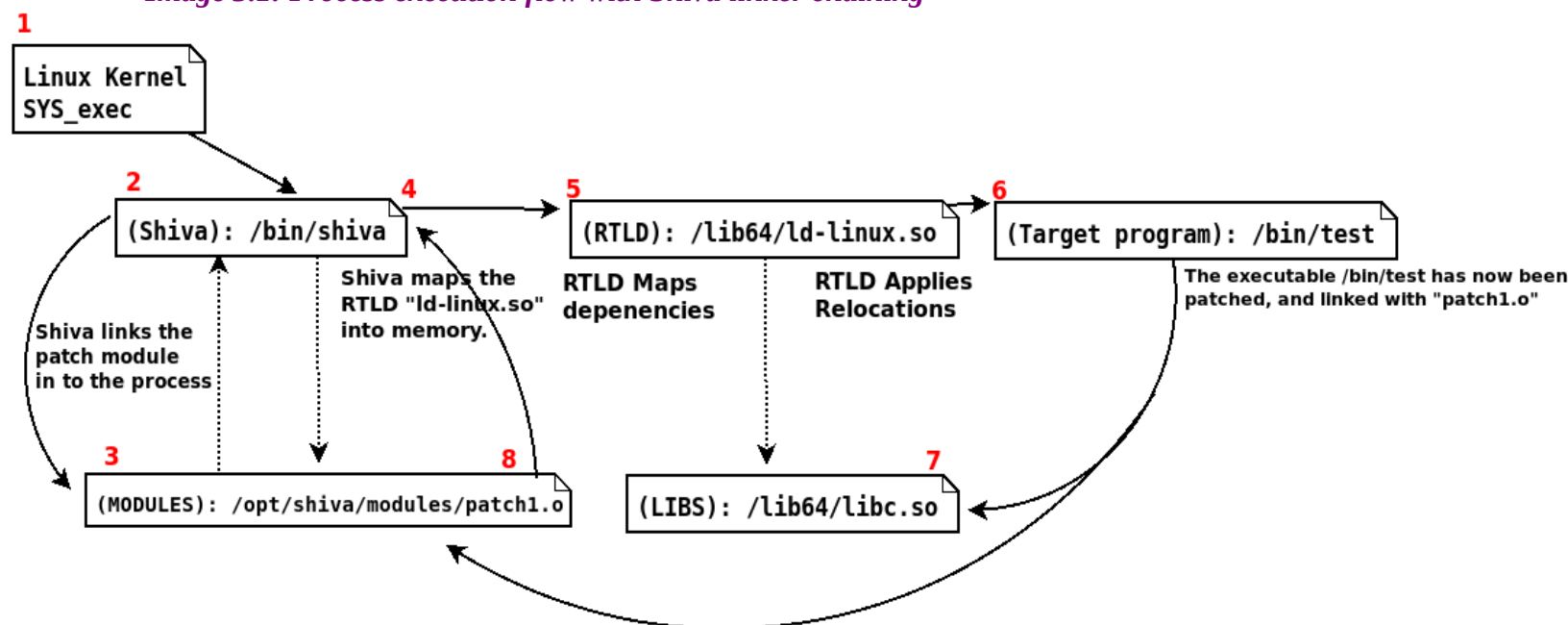
*Linux kernel code for mapping the program interpreter:*

[https://elixir.bootlin.com/linux/latest/source/fs/binfmt\\_elf.c#L592](https://elixir.bootlin.com/linux/latest/source/fs/binfmt_elf.c#L592)

Shiva also may execute directly as a regular static ELF executable, and it has a complete userland implementation of `execve` *See grugqs original work on userland exec [2]* that lets Shiva essentially be responsible for loading the entire address space at runtime. Shiva can be executed directly on the command line with the binary and patch object specified as arguments. In this case Shiva will load and map the target ELF program into memory along with it’s *Program interpreter* “/lib/ld-linux.so”, and once it has finished linking the required patch object into memory, the execution flow is passed to “/lib/ld-linux.so” so that it can finish loading and relocating the dependency modules.

Shiva is a *Weird Machine* in that it can be executed directly or indirectly offering flexibility in it’s use-cases. *See Shiva slide-deck [3]*

*Image 3.1: Process execution flow with Shiva linker chaining*



## 4.0 – Shiva runtime linking and loading of patches

The Shiva runtime linker aims to be programmable, flexible, and offer a relatively natural way to patch software within the context of the existing ELF ABI toolchain. The Shiva linker is a runtime linker, similar in nature to “/lib/ld-linux.so” in the sense that it performs ELF runtime linking, but akin too with the static ELF linker “/bin/ld” as Shiva uses ELF Relocatable objects for modules instead of ELF Shared libraries. Additionally, Shiva is programmable from a *Shiva patch module* and implements special directives for *program transformation* and *cross relocation* (See sections 4.7, 4.5).

Shiva uses ELF relocatable objects as a way to instrument symbolic patch code and data into the process image in the same sense that “/bin/ld” links an object into an ELF executable. The Shiva linker also aims to support custom ELF meta-data called *Transformation Records* that serve as instructions to Shiva’s program transformation engine and are defined within the patch code.

### 4.1 – Process address space requirements

The Shiva runtime environment is relatively complex in it’s fullness and many careful considerations have been taken while designing the runtime loading. A Shiva module (containing a patch) is an ELF relocatable object, a small file containing no loadable segments. Similar to how the Linux kernel must prepare an executable runtime environment for an LKM, we too must prepare an execution environment for the Shiva module. For this, Shiva creates a suitable text segment, data segment, .bss area, and copies the necessary code and data from the Shiva modules ELF sections over into the appropriate mapped segments, which are anonymous memory mappings.

Once the Shiva module containing all of the patch code has been loaded and linked into memory, the program that we are patching will be linked via *XREF’s (Branches to code, and data references)* to the Shiva modules patch code. In the event that the Linux kernel loads the Shiva module at an anonymous memory mapping that is more than 4GB in distance from the ELF executable we are linking it to, then the ELF relocations for the Shiva module won’t be able to encode large enough offsets. It’s important to mention that when Shiva is invoked directly (vs. as *an indirect program interpreter*) it is fully responsible for loading the module *and* the target executable into memory and therefore it can guarantee all modules will be within 4GB range of eachother. We must account for when Shiva is executed indirectly as a program interpreter though, which is it’s most common mode of operation for AMP, and we therefore defer to using a **large code model** when building the Shiva modules. This compiles code that uses PLT/GOT’s to handle code transfer in large code models.

The executable that is being patched generally has no ELF relocations beyond the normal *.rela.plt* and *.rela.dyn* sections which contain relocations utilized only by the “ld-linux.so” rtld. Therefore Shiva must perform live runtime analysis to acquire on-the-fly relocation information about where and how the target executable should be re-linked with the Shiva module at runtime. In the future we want our prelinking tool *shiva-ld* to emit pre-compiled relocation information about the target executable itself, that can be utilized by the Shiva linker at runtime.

If the process image requires a large code model, and our Shiva module’s are built with a large code model we have only solved half the problem; the ELF executable we are patching will almost always be compiled and linked with the code mechanics for a small code model. In this case there are two solutions:

1. Stick with a small code model, but with one big limitation. Execute Shiva directly (i.e. “/lib/shiva/path/to/executable”). When executing Shiva directly it can guarantee that all modules including the executable are mapped with MAP\_32BIT and will remain in a 4GB address space from each other.

2. Use a large code model when building Shiva modules, i.e. ‘`gcc -fno-pic -mcmodel=large -c patch.c`’. This will generate the correct relocations for a large code model, so that the module can be properly linked at runtime. In order to address the re-linking of the ELF executable which is likely built with a small code model we must instrument a ***jmp stub*** that bridges the gap between the executable code and Shiva module, which may likely be mapped further apart than can be addressed with 32bits.

We generally defer to the 2<sup>nd</sup> option as it is flexible and can handle all possible code models and it doesn’t matter whether Shiva is running in *Interpreter mode* or not.

Another little issue that arose in x86\_64 is that the Shiva interpreter when compiled as a no-pie static ELF executable with glibc, has initialization code that is not compatible with being executed indirectly, as it tries to incorrectly acquire the location of its own PT\_TLS segment from the program header table of the executable being loaded and not the interpreter program. This is caused by the fact that it retrieves the location of the program header table from the auxiliary vectors AT\_PHDR entry. To remedy this problem, Shiva is compiled and linked as a musl-libc static-pie executable using a custom musl patch, and avoids this problem completely.

Other interesting caveats and challenges are presented when building an intricate process transformation technology, but we will not include an exhaustive list beyond what is necessary to explain certain design decisions such as a large code model, and musl-libc.

## 4.2 – ELF ET\_REL modules for patching

A complex technology such as microcode patching at runtime requires granularity, flexibility, and precision. Relocatable code is designed specifically for the purpose of linking code and data together via symbolic references, and it provides enough *Relocation data* to instruct a linker to re-landscape a process image in very nuanced ways. Additionally, relocatable code doesn’t contain program segments and takes up less space on disk than a shared object file, and it allows Shiva to build an entirely custom runtime image for the module, one that is tailored towards the predilections of our overall patching environment. Furthermore relocatable object’s allow us to build a single runtime image (i.e. a text and data segment) from multiple patch files. If each patch file was a shared library, it would require mapping the segments of each shared library creating multiple executable images, requiring a lot more space and losing the inherent strengths of a single text and data segment housing multiple modules.

Currently, for Shiva-AArch64 the following ELF relocation types are supported, and are elected by the ELF linker in order to create code that supports a *large code model* (i.e. `gcc -mcmodel=large`).

### *AArch64 Relocations supported by Shiva*

<b>R_AARCH64_ABS64:</b>	$(S + A)$
<b>R_AARCH64_CALL26:</b>	$((S + A - P) \gg 2) \& 0x3ffffff$
<b>R_AARCH64_ADD_ABS_LO12_NC:</b>	$(S + A) \& 0xfff$

**R\_AARCH64\_ADR\_PREL\_PG\_HI21:**  $((\text{PAGE}(\text{S}+\text{A}) - \text{PAGE}(\text{P})) \gg 12) \& 0\text{x1ffff}$

These relocation types listed above are aarch64 ELF relocations for linking an object file into an executable image. In our case we link the object file into the executable memory image that is setup by Shiva for the loaded module. Sections such as the `.text`, `.rodata`, and `.plt` are placed within a mapped *Text segment*. The `.data`, `.got.plt`, and `.bss` sections are placed within a mapped *Data segment*.

A process image for a Shiva module can combine multiple patch objects into a single executable runtime image. The ELF patch object relocations are used specifically to fix up the *patch code* at runtime, but that does not solve the fact that the executable must be re-linked in memory still.

*NOTE: The executable must still be properly reconstructed and linked with the patch module code and patch data as necessary. ELF binaries do not contain the type of relocations that we are looking for. This posits the challenge of needing to generate relocation information on the fly, which is slow on large programs. A future version will use pre-compiled relocation data for the binary being patched. Even legacy binaries that are already built can have pre-compiled relocation data generated and stuffed into an ELF section.*

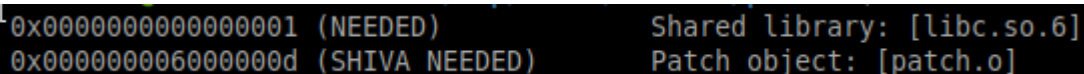
In addition to the standard ELF relocations, Shiva supports “transformations” a custom meta-data type that can be generated by *shiva-ld* (See section 4.7).

## 4.2.A DT\_NEEDED\_SHIVA entry in Dynamic segment

The dynamic segment is stored within the data segment, and contains various meta-data that is necessary for the dynamic linker “ld-linux.so” at runtime. The dynamic linker recognizes the *DT\_NEEDED* dynamic tag as a pointer to a dependency that should be loaded, such as *libc.so*. When we apply a patch with the *shiva-ld* tool, a custom dynamic segment entry is added to the binary. *DT\_SHIVA\_NEEDED* has a pointer to the path of a patch module that is to be loaded. The Shiva linker parses this dynamic tag in order to retrieve the path to the module that it’s loading.

```
#define DT_SHIVA_NEEDED 0x6000000d
```

*Image 4.2.1: Output from “readelf -d | grep NEEDED” might look like this*



```
0x0000000000000001 (NEEDED)           Shared library: [libc.so.6]
0x000000006000000d (SHIVA_NEEDED)      Patch object: [patch.o]
```

## 4.2.B Applying a patch with shiva-ld (The shiva pre-linker)

Up until this point we have discussed some of the linking and loading mechanics that are applied at runtime. We discussed using Shiva’s *ulexec* feature, which allows Shiva to run as a normal program that is fully responsible for loading and linking the ELF executable and ELF patch into memory. Shiva is also able to execute indirectly (*As an ELF Program interpreter loaded by the kernel*) when it is specified as the *PT\_INTERP* interpreter path, but can be heavily optimized in this case by embedding



pre-compiled Shiva-relocation data for program transformation and linking into the target executable's custom dynamic segment.

#### Image 4.2.2: An unpatched program

```
elfmaster@esoteric-aarch64:~/amp/shiva$ readelf -l test_function_patch | grep interpreter
[Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
elfmaster@esoteric-aarch64:~/amp/shiva$
elfmaster@esoteric-aarch64:~/amp/shiva$
elfmaster@esoteric-aarch64:~/amp/shiva$ ./test_function_patch
I'm the original foo() function!
The value of data_var is 5
I am a function that won't be patched
I'm accessing data var and its value is 0x5
```

As we can see in the unpatched `test_function_patch` program, the standard dynamic linker is in `PT_INTERP`, which indicates that this binary has not been statically modified to invoke Shiva as its interpreter. Nonetheless, Shiva can still patch the program entirely at runtime with no modifications to the executable if Shiva is invoked directly.

#### Image 4.2.3: Runtime only Patching with Shiva ulexec loading

```
elfmaster@esoteric-aarch64:~/amp/shiva$ set SHIVA_PATCH_SEARCH_PATH=./modules/patches/fpatch.o
elfmaster@esoteric-aarch64:~/amp/shiva$ ./shiva test_function_patch
I am the new function foo, and the new data_var is :0x31337
I am a function that won't be patched
I'm accessing data_var and its value is 0x31337
elfmaster@esoteric-aarch64:~/amp/shiva$
```

The example above (Image 4.2.3) shows how Shiva can be responsible for loading, patching, and executing the patched program at runtime. Shiva is being invoked directly, and therefore `test_function_patch` is only patched when running “`./shiva test_function_patch`”. If we were to run “`./test_function_patch`” alone, it would execute normally (Without the patch).

Most of the time when patching an executable, we want it to be more permanent. Having to run Shiva directly everytime you want to patch is not going to work. This is why Shiva runs predominately in Interpreter mode. Let's patch the program `test_function_patch` with the `shiva-ld` tool (The Shiva pre-linker).

#### Image 4.2.4: Shiva-ld applies “Interpreter Path” and “Module Path” to binary



```
elfmaster@esoteric-aarch64:~/amp/shiva$ shiva-ld test_function_patch /lib/shiva /opt/shiva/modules/fpatch.o
Done.
elfmaster@esoteric-aarch64:~/amp/shiva$ readelf -l test_function_patch | grep interpreter
[Requesting program interpreter: /lib/shiva]
elfmaster@esoteric-aarch64:~/amp/shiva$ ./test_function_patch
I am the new function foo, and the new data_var is :0x31337
I am a function that won't be patched
I'm accessing data_var and its value is 0x31337
elfmaster@esoteric-aarch64:~/amp/shiva$
```

The *shiva-ld* program applies several modifications to the *test\_function\_patch* ELF executable.

1. Specifies “/lib/shiva” as the interpreter in the PT\_INTERP segment
2. Creates a new dynamic segment area in the executable (*See 4.2.A on custom PT\_DYNAMIC*)
  - 2.A **SHIVA\_DT\_SEARCH** specifies the module search path “/opt/shiva/modules/”
  - 2.B **SHIVA\_DT\_NEEDED** specifies the module basename “fpatch.o”
  - 2.C **SHIVA\_DT\_EXTERNAL\_REL** location of pre-compiled reloc data for the executable
  - 2.D **SHIVA\_DT\_TRANSFORMS** location of pre-compiled *Transform records* (*See section 4.7*)

The *shiva-ld* tools is responsible for applying the correct ELF meta-data to the binary so that at runtime the kernel loads “/lib/shiva” as the interpreter, which in turn will load and link the patch. Since ELF executables don’t contain ELF relocation data (*Other than for shared library imports*) Shiva can and will analyze every instruction in the executable’s *.text* to find all instructions that must be linked to the patch module; this is slow at runtime, and can be heavily optimized by *shiva-ld* generating pre-compiled relocation data that lives in *.shiva.xref.reloc*. (*See section 5.0 Shiva runtime linking on target executable*). Additionally custom Shiva relocation records that are specifically for *Program Transformation*. To avoid all of the runtime calculations necessary for building up to *Program Transformation*, *shiva-ld* will generate the necessary transformation meta-data, and store it in *.shiva.ptd.reloc*.

The *shiva-ld* tool creates an extra loadable segment in the target executable, large enough for an updated *PT\_DYNAMIC* segment, and all of the linking meta-data for Shiva.

So compiling and installing a patch to an ELF binary happens in 3 steps:

#### *Pre-Runtime: Compile the patch, and pre-link it to the executable*

1. `gcc -fno-pic -mmodel=large fpatch.c -o fpatch.o`
2. `shiva-ld ./test /lib/shiva /opt/shiva/modules/fpatch.o`

#### *Runtime: The Shiva linker applies the patch at runtime*

3. `./test`

## 4.3 Symbol interposition patching

It is our goal to make patching as natural as possible. Ideally we would like to be able to write our entire patch in C, or some C like expression. Our current Shiva implementation expects that the

binaries it will be patching lack *DWARF* information, and therefore we can only use C code when it applies directly to patching code or data that is represented by a symbol of type *STT\_OBJECT* or *STT\_FUNC*; essentially global variables and functions. Replacing an entire function, or updating the value of a global variable is accomplished via *Symbol Interposition*.

### From “Oracle linkers and libraries guide”

*“Interposition can occur when multiple instances of a symbol, having the same name, exist in different dynamic objects that have been loaded into a process. Under the default search model, symbol references are bound to the first definition that is found in the series of dependencies that have been loaded. This first symbol is said to interpose on the other symbols of the same name.”*

With Shiva, we adopt the dynamic linkers concept of symbol interposition while breaking some of the traditional rules in order to give us more flexibility and power. Typically symbol interposition is primarily taken advantage of with the shared libraries that want to replace the code or data within a shared library of lesser precedence. Therefore it is limited to modifying functions or data within shared libraries only. Shiva allows for linking new functions and data into the executable program itself.

Let’s delve into an example of a simple program that we are going to patch.

### AArch64 ELF program *test\_p1.c*

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

int data_var = 0x1000;

int foo(void)
{
    printf("I'm the original foo(), and here's my data_var: %#lx\n", data_var);
    return 0;
}

int main(void)
{
    foo();
}
~
~
~
```

In this example we will show the source code just to make things extremely clear. The source code for *test\_p1* shows a *main()* function calling function *foo()*. In some cases it may be desirable to completely replace a function or global data (Such as *.data*, *.rodata*, or *.bss* variables). Essentially re-linking the process image to use the symbols/code/data created in a Shiva patch object.

### Truncated symbol table of ELF binary “test\_p1”

75: 00000000000011010	4	OBJECT	GLOBAL	DEFAULT	21	data_var
81: 0000000000000724	44	FUNC	GLOBAL	DEFAULT	13	foo
86: 0000000000000750	24	FUNC	GLOBAL	DEFAULT	13	main

We can the relevant symbols in this binary that will ultimately be relinked with the patch symbols. The STT\_FUNC symbol *foo* is STB\_GLOBAL. Therefore in our patch we will have to include the new function *foo* as an STB\_GLOBAL. If we plan to patch the value of the data segment variable *data\_var* we will have to include the new variable in our patch as an STT\_OBJECT with GLOBAL bindings.

### *Source code for patch “p1.c”*

```
int data_var = 0x31337;

int foo(void)
{
    printf("I am the new function foo, and the new data_var is :%#lx\n", data_var);
    return 0;
}
```

Our patch is written purely in C, and can be compiled as an ELF relocatable object. We must use a large code model (*See section 4.1*). Once the patch is applied, the executable will call the new version of *foo*, and reference the new version of *data\_var*. The new *foo*, and *data\_var* live within the text and data segment of the patch object’s executable runtime image.

*NOTE: It is possible for an interposing replacement function, such as *foo()*, to call back to the original *foo()*. Shiva has a symbol resolver *shiva\_resolve\_symbol()* that can resolve symbols within the target executable. This can be very desirable when wanting to insert a patch that sanitizes data or performs a needed check before calling the original *foo*. See section 4.7.C*

### *Symbol and relocation data for patch “p1.o”*

```
elfmaster@esoteric-aarch64:~/amp/shiva/modules/patches$ readelf -s p1.o

Symbol table '.symtab' contains 16 entries:
   Num:    Value              Size Type      Bind   Vis      Ndx Name
   ---:    ---              -
   0: 0000000000000000      0 NOTYPE   LOCAL  DEFAULT  UND
   1: 0000000000000000      0 FILE     LOCAL  DEFAULT  ABS p1.c
   2: 0000000000000000      0 SECTION  LOCAL  DEFAULT    1
   3: 0000000000000000      0 SECTION  LOCAL  DEFAULT    3
   4: 0000000000000000      0 SECTION  LOCAL  DEFAULT    4
   5: 0000000000000000      0 NOTYPE   LOCAL  DEFAULT    3 $d
   6: 0000000000000000      0 SECTION  LOCAL  DEFAULT    5
   7: 0000000000000000      0 NOTYPE   LOCAL  DEFAULT    5 $d
   8: 0000000000000000      0 NOTYPE   LOCAL  DEFAULT    1 $x
   9: 00000000000000038      0 NOTYPE   LOCAL  DEFAULT    1 $d
  10: 00000000000000040      0 NOTYPE   LOCAL  DEFAULT    1 $d
  11: 0000000000000000      0 SECTION  LOCAL  DEFAULT    7
  12: 0000000000000000      0 SECTION  LOCAL  DEFAULT    6
  13: 0000000000000000      4 OBJECT   GLOBAL  DEFAULT    3 data_var
  14: 0000000000000000      52 FUNC     GLOBAL  DEFAULT    1 foo
  15: 0000000000000000      0 NOTYPE   GLOBAL  DEFAULT  UND printf

elfmaster@esoteric-aarch64:~/amp/shiva/modules/patches$ readelf -r p1.o

Relocation section '.rela.text' at offset 0x2a0 contains 7 entries:
   Offset             Info             Type             Sym. Value      Sym. Name + Addend
   ---:             ---:             ---:             ---:             ---:
0000000000000008 000200000113 R_AARCH64_ADR_PRE 0000000000000000 .text + 38
000000000000000c 000200000115 R_AARCH64_ADD_ABS 0000000000000000 .text + 38
0000000000000018 000200000113 R_AARCH64_ADR_PRE 0000000000000000 .text + 40
000000000000001c 000200000115 R_AARCH64_ADD_ABS 0000000000000000 .text + 40
0000000000000024 000f0000011b R_AARCH64_CALL26 0000000000000000 printf + 0
0000000000000038 000d00000101 R_AARCH64_ABS64   0000000000000000 data_var + 0
0000000000000040 000600000101 R_AARCH64_ABS64   0000000000000000 .rodata + 0

elfmaster@esoteric-aarch64:~/amp/shiva/modules/patches$
```

The symbol and relocation data for p1.o are all of the ELF meta-data necessary for the Shiva linker to load and link the module into a process address space. These relocations are only responsible for linking the code in the module into the address space; external linking to the executable is required to complete linking to the patch code (*See section 5.0: Shiva runtime re-linking on executable*).

After the p1.o patch has been fully linked into the running program and the external linking has been applied to the executable, every call to *foo()* and every reference to the *data\_var* variable within the executable will be linked to the updated versions of the symbol from the p1.o patch, containing the new *foo()* and *data\_var*.

***Running the ./test\_p1 program unpatched and then patched***

```
elfmaster@esoteric-aarch64:~/amp/shiva$ readelf -l test_p1 | grep interpreter
[Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
elfmaster@esoteric-aarch64:~/amp/shiva$ ./test_p1
I'm the original foo(), and here's my data_var: 0x1000
elfmaster@esoteric-aarch64:~/amp/shiva$ tools/add_elf_interp64 test_p1 /lib/shiva
Done.
elfmaster@esoteric-aarch64:~/amp/shiva$ readelf -l test_p1 | grep interpreter
[Requesting program interpreter: /lib/shiva]
elfmaster@esoteric-aarch64:~/amp/shiva$ ./test_p1
I am the new function foo, and the new data_var is :0x31337
elfmaster@esoteric-aarch64:~/amp/shiva$
```

The output above is demonstrating how the program `./test_p1` has the path to the standard dynamic linker, in this case “`/lib/ld-linux-aarch64.so.1`”. When we execute the program it behaves as expected, before the patch. We use `add_elf_interp64` to change the `PT_INTERP` segment path to “`/lib/shiva`”. The patch object: `p1.o`, is being loaded from a fixed path. Shiva is still a prototype and only loads one patch object from the fixed path “`./modules/shakti_runtime.o`”, in the near future a module search path and module basenames will be included in the custom dynamic segment to direct Shiva where to load the module from.

Shiva ignores the normal symbol visibility rules of symbol interposition in order to aggressively patch inlined functions or hidden symbols (i.e. `STV_HIDDEN`). The above example shows a more traditional example of overriding a function with `STB_GLOBAL` bindings and `STV_DEFAULT` visibility.

## 4.5 Cross ELF relocations between chained linkers

Cross ELF relocation occurs when a symbol reference from within a patch object is referencing a symbol or function that is meant to be relocated by the “`ld-linux.so`” `rtld`. In example... Shiva is loading and linking the patch object into memory, and the patch is calling function `printf` that lives in `libc.so`. The dynamic linker “`ld-linux.so`” hasn’t even begun execution yet, let alone resolution of it’s shared library symbols, so how does Shiva fixup the `R_AARCH64_CALL26` relocation that binds the call instruction to `libc.so:printf`? This is a kunundrum where two runtime linkers that load a totally separate set of modules, must synchronize relocation data in some way. Shiva calls this *Cross ELF relocation*.

Let’s take a look at the ELF relocations and symbols for the `p1.o` patch object, one more time.

## Symbol and relocation data for patch “p1.o”

```
elfmaster@esoteric-aarch64:~/amp/shiva/modules/patches$ readelf -s p1.o

Symbol table '.symtab' contains 16 entries:
   Num:      Value              Size Type      Bind     Vis      Ndx Name
   ---:      -
    0: 0000000000000000         0 NOTYPE   LOCAL   DEFAULT  UND
    1: 0000000000000000         0 FILE    LOCAL   DEFAULT  ABS p1.c
    2: 0000000000000000         0 SECTION LOCAL   DEFAULT    1
    3: 0000000000000000         0 SECTION LOCAL   DEFAULT    3
    4: 0000000000000000         0 SECTION LOCAL   DEFAULT    4
    5: 0000000000000000         0 NOTYPE   LOCAL   DEFAULT    3 $d
    6: 0000000000000000         0 SECTION LOCAL   DEFAULT    5
    7: 0000000000000000         0 NOTYPE   LOCAL   DEFAULT    5 $d
    8: 0000000000000000         0 NOTYPE   LOCAL   DEFAULT    1 $x
    9: 00000000000000038         0 NOTYPE   LOCAL   DEFAULT    1 $d
   10: 00000000000000040         0 NOTYPE   LOCAL   DEFAULT    1 $d
   11: 0000000000000000         0 SECTION LOCAL   DEFAULT    7
   12: 0000000000000000         0 SECTION LOCAL   DEFAULT    6
   13: 0000000000000000         4 OBJECT   GLOBAL   DEFAULT    3 data_var
   14: 0000000000000000        52 FUNC     GLOBAL   DEFAULT    1 foo
   15: 0000000000000000         0 NOTYPE   GLOBAL   DEFAULT  UND printf

elfmaster@esoteric-aarch64:~/amp/shiva/modules/patches$ readelf -r p1.o

Relocation section '.rela.text' at offset 0x2a0 contains 7 entries:
   Offset             Info             Type             Sym. Value      Sym. Name + Addend
   ---:             -
0000000000000008 000200000113 R_AARCH64_ADR_PRE 0000000000000000 .text + 38
000000000000000c 000200000115 R_AARCH64_ADD_ABS 0000000000000000 .text + 38
0000000000000018 000200000113 R_AARCH64_ADR_PRE 0000000000000000 .text + 40
000000000000001c 000200000115 R_AARCH64_ADD_ABS 0000000000000000 .text + 40
0000000000000024 000f0000011b R_AARCH64_CALL26 0000000000000000 printf + 0
0000000000000038 000d00000101 R_AARCH64_ABS64 0000000000000000 data_var + 0
0000000000000040 000600000101 R_AARCH64_ABS64 0000000000000000 .rodata + 0

elfmaster@esoteric-aarch64:~/amp/shiva/modules/patches$
```

The symbol table contains *printf* as a STT\_NOTYPE symbol type, with an STT\_GLOBAL symbol binding. This tells Shiva that the symbol does not exist within the patch module, but somewhere external. The R\_AARCH64\_CALL26 relocation instructs Shiva to insert the encoded 26bit offset to *printf* within the *bl* instruction at offset 24.

```
modules/patches/p1.o:      file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <foo>:
  0: a9bf7bfd      stp     x29, x30, [sp, #-16]!
  4: 910003fd      mov     x29, sp
  8: 90000000      adrp    x0, 0 <foo>
  c: 91000000      add     x0, x0, #0x0
 10: f9400000      ldr     x0, [x0]
 14: b9400001      ldr     w1, [x0]
 18: 90000000      adrp    x0, 0 <foo>
 1c: 91000000      add     x0, x0, #0x0
 20: f9400000      ldr     x0, [x0]
 24: 94000000      bl      0 <printf>
 28: 52800000      mov     w0, #0x0                                // #0
 2c: a8c17bfd      ldp     x29, x30, [sp], #16
 30: d65f03c0      ret
 34: d503201f      nop
  ...
```



Shiva must locate the *printf* symbol in a module outside of the patch object, and relocate the patch object's *bl* instruction accordingly.

### *Steps to linking an external function, with Cross Relocation*

1. Check to see if *printf* lives within the target executable. If it exists in the target executable, and is STT\_GLOBAL/STT\_FUNC, then Shiva will link it to the *bl* instruction by solving the R\_AARCH64\_CALL26 relocation. Otherwise Shiva moves on to step 2.
2. Check if *printf* is in the target executable symbol table with STT\_NOTYPE, which indicates that the symbol is external, and will likely be linked at runtime. If so, then Shiva checks to see if there exists a PLT entry for calls to *printf*. If a PLT doesn't exist, then linking fails. Otherwise, Shiva resolves the R\_AARCH64\_CALL26 relocation so that it contains the offset to the *printf@PLT* entry within the target executable, in this way all calls to *printf* from the patch module will be transferred through the existing PLT entry. If lazy linking is used, the PLT/GOT entry won't be resolved until the *chained linker* "ld-linux.so" is invoked and finishes its dynamic fixups. If *printf* doesn't exist at all in the symbol table of the target executable, then move to step 3.
3. If we have arrived at this step, then Shiva could not find the *printf* symbol in the target executable's symbol table, even as an undefined symbol it is not present. In this case we search all of the shared library dependencies, transitively, via the *DT\_NEEDED* entries in the dynamic segment of the executable and it's shared modules. If the symbol doesn't exist then Shiva exits with a linker failure, as it cannot be resolved. If Shiva finds the symbol *printf* as STB\_GLOBAL/STT\_FUNC within a shared object dependency (Such as libc.so), then Shiva must *generate new relocation tables* containing a new R\_AARCH64\_JUMP\_SLOT relocation, and a new PLT stub that support the linking of *printf* at runtime.

### *Steps to generating a new JUMP\_SLOT relocation*

1. Shiva creates an alternate *.rela.plt* section, that lives in memory allocated by *malloc(3)*. Copies the original *.rela.plt* into the new location, and adds an extra R\_AARCH64\_JUMPSLOT relocation at the end of the *.rela.plt* table. The *r\_offset* field should point to the address of the new *.got.plt* entry created in step 2.
2. Create a custom *.got.plt* entry, and a *.plt* stub specifically for the function that you are wanting to resolve at runtime. Each entry is a pointer-width in size, and should be allocated in the heap via *malloc(1)*.
3. Create a new symbol table, containing the original symbol data and the symbol for the new function you are creating the entry for. This symbol table can be copied into memory allocated by *malloc(3)*.
4. Update *DT\_JMPREL* (in the dynamic segment) to point to our new *.rela.plt* table
5. Update *DT\_JMPRELSZ* to contain the size of our newly created relocation table (original\_size\_of\_table + sizeof(Elf64\_Rela)).
3. Link all module call instructions with the offset that points to the newly added PLT stub.



This is a great example of how the first linker “/lib/shiva” can interact with and even instruct it’s chained linker “/lib/ld-linux.so” by generating ELF meta-data that “ld-linux.so” understands, specifically relocation data for linking purposes.

*NOTE: In the event that we were linking to a variable instead of a function, Shiva would generate an `R_AARCH64_GLOB_DAT` relocation instead of an `R_AARCH64_JUMP_SLOT`.*

## 4.6 Declaring external variables within patch source

The last section (4.5 *ELF Cross relocation between linkers*) gave an example of linking to external code and data implicitly, leading to cross-relocation. In many cases though too, it is desirable to simply declare an external variable, without patching it. Let’s take a look at an example program, and a corresponding patch to illustrate our needs.

*Source code for test\_p2.c, a program we are patching*

```
char bss_buffer[16];

void print_buffer(void)
{
    printf("%s\n", bss_buffer);
}

int main(int argc, char **argv)
{
    if (argc > 2) {
        strncpy(bss_buffer, argv[1], sizeof(bss_buffer) - 1);
        print_buffer();
    }
}
```

This program simply copies the `argv[1]` into the `bss_buffer`, and prints the string with `printf`. Though, let’s write a simple patch that only prints the buffer when it contains the string “hello world”.

*Source code for p2.c patch*

```
extern char bss_buffer[16];

void print_buffer(void)
{
    if (strcmp(bss_buffer, "hello world") == 0)
        printf("%s\n", bss_buffer);
}
```

Notice that our patch code declares *bss\_buffer* using the *extern* keyword. Because our patch wants to read from the *bss\_buffer* from the executable vs. creating a new *bss\_buffer*. This results in the symbol type being *STT\_NOTYPE* vs. *STT\_OBJECT*. If we fail to apply the *extern* keyword, Shiva would attempt to interpose the symbol *bss\_buffer* and all of the program that references *bss\_buffer* would be relinked to use the patches version of *bss\_buffer*.

When Shiva applies the patch above, the program will only print *argv[1]* to stdout if it is the string “hello world”.

## 4.6 Program Transformations (Transforms)

Although symbol interposing is a strength in that it’s a natural grammar for re-writing and re-linking code and data that are represented by ELF symbols, it cannot readily be applied to the byte-code sequences in between all of the symbol definitions. For example, adding several bytes of code to an existing function requires that the function be extended in length. Let’s say we want to add 8 bytes in the middle of a function, this would cause any instructions that live after the insertion to have invalid *xref* offsets for any calls or references to code and data that live before the insertion. Therefore a complete *transformation* of the function being patched is required. This is where *Transforms* come into play. They give the patch writer the mechanics necessary to handle patching more nuanced parts of the program code. Shiva implements these directives as macros, that are interpreted by the *Shiva Pre-linker*, which interprets them and compiles them into custom Shiva Transformation records, similar to Relocation records.

Consider a simple function that has a minor security vulnerability due to *strcpy* usage. We must write a patch to fix this security vulnerability, and in doing so we will demonstrate how *Transforms* can be used to directly inform Shiva on where to emit code, and Shiva will handle all of the transformation under the hood.

*Image 4.6.1: Vulnerable function source code strcpy\_vuln.c*

```
#define MAX_BUF_LEN 32

void vuln(char *string)
{
    char buf[MAX_BUF_LEN];

    strcpy(buf, string);
    printf("buf: %s\n", buf);
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

Our goal is to patch the function *vuln* directly, so that it uses *strncpy* instead of *strcpy*. Now since this *vuln* function is so simple, we could simply re-write it with a new version and patch it with *symbol*

*interposition*, but in real life scenarios the function is often too complex to re-write, especially without source code. Let's take a look at what the patched assembly for the function *foo* needs to look like

**Image 4.6.2: Assembly diff for patched function vuln**

```
0000000000000000 <vuln>:
0x400000    stp    x29, x30, [sp, #-64]!
0x400004    mov    x29, sp
0x400008    str    x0, [x29, #24]
0x40000c    add    x0, x29, #0x20
0x400010 + mov    x2, #31
0x400014    ldr    x1, [x29, #24]
- bl        0 <strcpy>
0x400018 + bl        0 <strncpy>
0x400020    add    x1, x29, #0x20
0x400024    adrp   x0, 0 <foo>
0x400028    add    x0, x0, #0x0
0x40002c    bl      0 <printf>
0x400030    nop
0x400034    ldp    x29, x30, [sp], #64
0x400038    ret
```

1. An instruction is added “*mov x2, #31*” at address *0x400010*, which is where the “*ldr x1, [x29, #24]*” was at, and has now been shifted forward to *0x400014* to make room.

2. The call instruction at *0x400018* “*bl strcpy*” has been replaced with “*bl strncpy*”.

These types of code insertions may appear simple, but they require a relatively robust system for program transformation at runtime. Inserting new code into a function that extends the size of the function will cause erroneous code after the insertion due to the extension changing *xref* offsets. Additionally, the call to *strcpy* has been changed to *strncpy* which may be tricky to link if there is no PLT stub in the executable.

**Image 4.6.3: Assembly code for original vuln functions**

```
0000000000000000 <vuln>:
0x400000    stp    x29, x30, [sp, #-64]!
0x400004    mov    x29, sp
0x400008    str    x0, [x29, #24]
0x40000c    add    x0, x29, #0x20
0x400010    ldr    x1, [x29, #24]
0x400014    bl      0 <strcpy>
0x400018    add    x1, x29, #0x20
0x40001c    adrp   x0, 0 <foo>
0x400020    add    x0, x0, #0x0
0x400024    bl      0 <printf>
0x400028    nop
0x40002c    ldp    x29, x30, [sp], #64
0x400030    ret
```

The above disassembly shows the function *vuln* before it has the patch applied. Our goal is to add one new instruction, and patch an existing *bl* instruction.

## 4.7 Patching manually with Transformation directives (The hard way)

1. To insert a new *mov* instruction in between the *add* at *0x40000c* and the *ldr* at *0x400010*, we can use the Shiva Transform macros for emitting new instructions, in this case a *mov* instruction.

To add the instruction “*mov x2, #31*”

We add the following line to our patch source code:

```
SHIVA_T_EMIT_NEW_MOV(0x400010, SHIVA_AARCH64_X2, 31);
```

*Defined as*

```
#define SHIVA_T_EMIT_NEW_MOV(address_of_insert, reg, imm)
```

The Shiva Pre-linker will interpret this command from the patch object, and compile it into *Transformation record* that is specific to the Shiva runtime, and specifically program transformation. To be more specific, this transform will be processed by the *Shiva Pre-linker* when applying the patch, and the custom transform data will be stored in the ELF section *.shiva.transform*. Therefore at runtime the Shiva linker only needs to read pre-processed meta-data to transform the function, which is relatively fast. If Shiva is being invoked directly on an executable that hasn't had the transformation/relocation meta-data installed (i.e. “*shiva p ./patch.o test\_prog*” vs. “*./test\_prog*”) then Shiva will have to calculate all of the transformation and relocation on the fly as it statically analyzes the code at runtime prior to linking.

A Transform directive that emits a new instruction will therefore extend the size of the function, and therefore require complete function relocation. The Shiva linker will make a copy of the old function moving it over to the Shiva modules memory space, insert any new instructions (Such as *mov*) and then handle any fixups in the code that live before or after the new instruction. Shiva will then fixup any calls or references to this function from within the target executable.

The *SHIVA\_T\_EMIT\_NEW\_<INSN>* macros will insert a new instruction in between two instructions, but sometimes we may want to simply overwrite an existing instruction. In this case we would use the *SHIVA\_T\_OVERWRITE\_<INSN>* which will overwrite an existing instruction, and is a lot less labor intensive since it isn't required to patch up relocations.

2. To modify the *bl* instruction from calling *strcpy* to calling *strncpy* we can add the following line to our patch source code:

```
SHIVA_T_EMIT_BL_RESOLVE(0x400014, “strncpy”);
```

### Defined as

```
#define SHIVA_T_EMIT_BL_RESOLVE(address_of_insert, symbol_name)
```

If we knew what the address of *strcpy* was going to be (*Which we won't during patch creation*) we could use

```
SHIVA_T_OVERWRITE_BL(0x400014, <offset>);
```

Often times when writing the patch though we may be linking to a symbol whose value we won't know until it's been linked by the "ld-linux.so" RTLD, therefore it is very friendly to have a *Resolver* version of certain Transforms to handle the semantics of *Cross Relocation* symbol resolution.

There exist *Transform directives* to emit (insert or overwrite) any ARM instruction. They are a work in progress as the design of Shiva continues. More of them will exist down the road for various arbitrary operations.

## 4.8 A natural C approach, with Shiva's *function splicing*

In many patching scenarios Shiva allows the patch to be designed in C, in the programming customs already known to developers.

Instead of using Shiva's Transform directives to emit instruction encodings, we can simply splice in the call to *strcpy* so that it overwrites *strcpy*, using the function splicing transformation *SHIVA\_T\_SPLICE\_FUNCTION*.

The patch identifies the name of the function *vuln()* that is to be transformed. The starting address of the splice insert, and the ending address. The patch also creates two .bss variables *buf*, and *string*. At runtime extra byte-code will be emitted that loads *char buf[MAX\_BUF\_LEN]* with the value at [*sp* + #32], and *char \*string* will be assigned the pointer value from register *x0*. All of this data will be encoded into the ET\_REL file (The patch object) in a way that Shiva can interpret and turn into a function splicing operation.

*Image 4.8.1: Example of function splicing in strcpy patch. (strcpy\_vuln\_patch.c)*

```

#define MAX_BUF_LEN 32

char buf[MAX_BUF_LEN];
char *string;

SHIVA_T_SPLICE_FUNCTION(vuln, 0x400008, 0x400018)
{
    SHIVA_T_LDV_FROM_MEM(buf, AARCH64_SP, +32);
    SHIVA_T_LDV_FROM_REG(string, AARCH64_X0);

    /*
     * Call to strncpy (Replacing strcpy)
     */
    strncpy(buf, string, MAX_LEN - 1);
}
~
~

```

In section 4.2.B we discussed the Shiva Pre-linker

which compiles these Transformations into a meta-data known as *transform-records* that can be stored within the executable that is being patched. Here is the preliminary definitions, with an example of applying one of these relocation fields at runtime.

#### Image 4.8.2: Shiva transform structs

```

typedef enum shiva_transform_type {
    SHIVA_TRANSFORM_SPLICE_FUNCTION = 0,
    SHIVA_TRANSFORM_EMIT_BYTECODE = 1,
    SHIVA_TRANSFORM_UNKNOWN
} shiva_transform_type_t;

typedef struct shiva_transform {
    shiva_transform_type_t type;
    struct elf_symbol target_symbol;
    struct elf_symbol source_symbol;
    struct elf_symbol next_func; /* symbol of next function after the one being spliced */
    uint64_t offset; /* offset initial transformation */
    uint64_t old_len; /* length of old code/data being inserted */
    uint64_t new_len; /* length of new code/data being inserted */
#define SHIVA_TRANSFORM_F_REPLACE (1UL << 0)
#define SHIVA_TRANSFORM_F_INJECT (1UL << 1)
#define SHIVA_TRANSFORM_F_NOP_PAD (1UL << 2)
#define SHIVA_TRANSFORM_F_EXTEND (1UL << 3)
    uint64_t flags; /* flags describe behavior, such as overwrite, extend, etc. */
    uint8_t *ptr; /* points to the new code or data that is apart of the transform */
    char *name; /* simply points to target_symbol.name */
    size_t segment_offset;
    TAILQ_HEAD(, shiva_branch_site) branch_list;
    TAILQ_HEAD(, shiva_xref_site) xref_list;
    TAILQ_ENTRY(shiva_transform) _linkage;
} shiva_transform_t;

```

The patching transformation records are stored into the patch modules *.shiva.transform* section, and at runtime the *Shiva Transformation Engine* will parse and apply the transformation. In the future the *shiva-ld* tool will store both *ELF Transformation* and *ELF Relocation* meta-data within the target-executable with pre-compiled analysis data etc. to speed up the transformation and runtime relinking process at initial load-time of Shiva.

*Image 4.8.3: Example code of how to apply a Transform relocation*

```
int apply_ptd_reloc(elfobj_t *elfobj, struct shiva64_ptd_rel *ptd_rel, uint8_t *new_func, uint8_t *old_func)
{
    switch(ptd_rel[i].type) {
    case SHIVA_PTD_R_TEXT_EMIT_INSN:
        res = copy_first_half(new_func, old_func, ptd_rel[i].transform_offset);
        res = insert_bytes(new_func, &ptd_rel[i]);
        res = copy_last_half(new_func, old_func, ptd_rel[i].transform_offset);
        res = relink_function_locals(new_func, ptd_rel[i].transform_offset);
        res = relink_xrefs_to_function(elfobj, &ptd_rel[i]);
        if (res == false) {
            fprintf("Failed to transform code in %s\n", ptd_rel[i].symbol.name);
            return false;
        }
        break;
    ...
}
```

## 5.0 Shiva runtime linking on the target executable

ELF executable's are not linked with enough relocatable data to link their code with the patch at runtime. A typical dynamically linked ELF executable usually contain 3 relocation types.

### *Relocations for “ld-linux.so”*

**R\_AARCH64\_RELATIVE**  
**R\_AARCH64\_JUMP\_SLOT**  
**R\_AARCH64\_GLOB\_DAT**

These relocations tells the “ld-linux.so” where to patch various writable data segment entries, such as the *.got*, *.got.plt*, *.init\_array*, and *.fini\_array*. These relocations are perfect for what the “ld-linux.so” needs in order to link in shared object dependencies, but there are not descriptive enough to help Shiva fixup the executable image so that it is linked with the patch code and data.

Shiva must analyze the executable code in the *.text* section at runtime, in order to disassemble every instruction, and find the various *xrefs* (*branches*, *memory references*, *etc.*) that will require being patched with the correct linking offsets. This can happen during one of two possible phases.

1. After Shiva loads and relocates the module it performs runtime analysis on the target executable in order to find all instructions that must be re-linked.
2. Before runtime, when *shiva-ld* is used to apply the patch, it will statically analyze the executable's *.text* section and generate ELF relocation data to describe the areas which must be patched within the



ELF executable. For example, for every *bl* instruction there would be an *R\_AARCH64\_CALL26* relocation.

The first option will certainly trigger a performance impact that increases based on the size of the *.text* section of the executable we are patching. Every instruction is disassembled on the fly at runtime, and then patched. This can be negligible on smaller programs or if the program load-time doesn't matter. When applying a patch to an executable with the *shiva-ld* (Shiva prelinker) the relocation data is generated and stored within the executable, so that everytime it is executed the Shiva linker can read the relocation/transform data and apply the patches to the executable so that it links with the patch code.

Whether the analysis happens during pre-linking or runtime, it always follows the same process to acquire instruction data and generate relocation information.

**Image 5.1: *adrp*/*add* instructions to load address of *&data\_var* from *.data* section**

```
00000000000007ac <bar>:
7ac: a9bf7bfd      stp     x29, x30, [sp, #-16]!
7b0: 910003fd      mov     x29, sp
7b4: 90000000      adrp    x0, 0 <_init-0x5d0>
7b8: 9123a000      add     x0, x0, #0x8e8
7bc: 97ffffa5      bl      650 <puts@plt>
7c0: b0000080      adrp    x0, 11000 <_data_start>
7c4: 91004000      add     x0, x0, #0x10
7c8: b9400001      ldr     w1, [x0]
7cc: 90000000      adrp    x0, 0 <_init-0x5d0>
7d0: 91244000      add     x0, x0, #0x910
7d4: 97ffffa3      bl      660 <printf@plt>
7d8: 52800000      mov     w0, #0x0 // #
7dc: a8c17bfd      ldp     x29, x30, [sp], #16
7e0: d65f03c0      ret
```

The *adrp*, and *add* instructions are generally used to reference a memory location, and therefore must be patched in order to re-link with the patch code and data. In the example above, the address value of *&data\_var* (A *.data* section variable) must be re-linked by patching the *adrp* instruction with the correct offset to the *patch objects* data segment, which will be within 4GB of the *adrp* instruction. The *add* instruction is patched with the correct offset of the new *data\_var* variable from the beginning of the data segment (where the modules respective *.data* section is).

The data segment for a patch is often loaded at *0x6000100*, and so we would have to first re-encode *adrp* with the correct offset encoded into 26 bits. Then handle the offset in *add* so that it contains the correct offset to the *data\_var* variable.

Currently Shiva re-links the following instruction types:

*As discussed in 4.2.B Shiva-Prelinker, this information will be generated into pre-compiled relocation data by shiva-ld when used, and stored within a new .shiva.ptd.rel.text or .shiva.ptd.rel.data.*

**Instruction:** *BL (Branch with link): To re-link calls from old function in executable to new function within patch module.*

**Generated Relocation Types:** *R\_AARCH64\_CALL26*

**NOTE:** *In the event of function pointers, we do not overwrite them, but do insert a trampoline in the old function so that it points to the new function. Therefore only function pointers will incur the overhead of an extra branch.*

**Instruction pair:** *ADRP/ADD (Page addr + offset) + Add-imm-offset: Patched to re-link with the new variable within the patch module.*

**Generated Relocation Types:** *AARCH64\_ADR\_PREL\_PG\_HI21,  
AARCH64\_ADD\_ABS\_LO12\_NC*

**Instruction pair:** *ADRP/LDR (Page addr + offset) + Ldr-imm-offset: Patched to re-link with the new variable within the patch module.*

**Generated Relocation Types:** *AARCH64\_ADR\_PREL\_PG\_HI21,  
AARCH64\_ADD\_ABS\_LO12\_NC*

More **Branch/Xref** instruction types will be addressed in future development.

## References:

1. Preloading the linker for fun and profit: <https://tmpout.sh/2/6.html>
2. Design and implementation of userland exec: [https://grugq.github.io/docs/ul\\_exec.txt](https://grugq.github.io/docs/ul_exec.txt)
3. Slide-Deck for Original x86\_64 Shiva implementation:  
[https://github.com/advanced-microcode-patching/shiva/blob/main/presentation/shiva\\_runtime.pdf](https://github.com/advanced-microcode-patching/shiva/blob/main/presentation/shiva_runtime.pdf)