Improving Disassembly and Decompilation

or

Moderately Advanced Ghidra Usage

Table of Contents

Intro and Setup

Improving Disassembly

Improving Decompilation: Data Types

Improving Decompilation: Function Calls

Improving Decompilation: Control Flow

Table of Contents

Improving Decompilation: Data Mutability

Improving Decompilation: Setting Register Values

Troubleshooting Decompilation

Contents

Intro and Setup
    Introduction
    Setup

Intro

- Like any SRE tool, Ghidra makes assumptions which sometimes need to be adjusted by reverse engineers.
- These slides describe techniques for recognizing problematic situations and steps you can take to improve Ghidra's analysis.
- These slides assume basic familiarity with Ghidra.
- Note: the materials for the "Beginner" and "Intermediate" Ghidra classes are included with the Ghidra distribution.

Setup

- First, create a new project for the example files used by these slides.
- Next, import the files. They are located in ⟨**ghidra_dir**⟩**/docs/GhidraClass/ExerciseFiles/Advanced**
- The easiest way to do this is to use the Batch Importer (**File** → **Batch Import...** from the Project Window).

Contents

Improving Disassembly
- Evaluating Analysis: The Entropy and Overview Windows
- Non-Returning Functions
- Function Start Patterns

Evaluation

- Use the entropy and overview sidebars to get a quick sense of how well a binary has been analyzed/disassembled.
- For instance, the entropy sidebar can tell you whether your binary has regions which are likely encrypted or compressed.
- To activate these sidebars, use the dropdown menu in the Listing (immediately to the right of the camera icon).

Non-returning Functions

- Some functions, like **exit** or **abort**, are **non-returning functions**. Such functions do not return to the caller after executing. Instead, they do drastic things like halting the execution of the program.
- Suppose **panic** is a function that does not return. The compiler is free to put whatever it wants (e.g., data) after calls to **panic**.
- If Ghidra does not know that **panic** is non-returning, it will assume that bytes after calls to **panic** are instructions and attempt to disassemble them.

Non-returning Functions

- The **Non-Returning Functions - Known** analyzer recognizes a number of standard non-returning functions by name and automatically handles them correctly.
- The **Non-Returning Functions - Discovered** analyzer attempts to discover non-returning functions by gathering evidence during disassembly.
- If a non-returning function manages to slip by these analyzers, it can wreak havoc on analysis. Fortunately, there are ways to recognize and fix this situation.

Exercise: Non-returning Functions

1. Open and analyze the file **noReturn**. Note: for all exercises, use the default analyzers unless otherwise specified.
2. Open the **Bookmarks** window and examine the **Error** bookmarks. There should be two errors.
3. These errors are due to one non-returning function that Ghidra doesn't know about. Identify this function and mark it as non-returning (right-click on the name of the function in the decompiler, select **Edit Function Signature** and select the **No Return** box).
4. Verify that the errors are corrected after marking the function as non-returning.

Exercise: Non-returning Functions

(advance for solutions)

Exercise: Non-returning Functions

(advance for solutions)

- The function **loopForever** is non-returning.
- Note: You can configure how much evidence the **Non-Returning Functions - Discovered** analyzer requires before deciding that function is non-returning via **Analysis** → **Auto Analyze ...** from the Code Browser. If you lower the evidence threshold, this analyzer will mark **loopForever** as non-returning.
- Also, the script **FixupNoReturnFunctions.java** will analyze a program and present a list of potentially non-returning functions. It will also allow you to mark a function as non-returning and repair any damage.

Finding Functions

- Ghidra uses many techniques to find bytes to disassemble and to group instructions together into function bodies.
- One such technique is to search for **function start patterns**. These are patterns of bits (with wildcards allowed) that indicate that a particular address is likely the start of a function.
- These patterns are based on two facts:
  1. Functions often start in similar ways (e.g., setting up the stack pointer, saving callee-saved registers)
  2. Similar things occur immediately before a function start (return of previous function, padding bytes,...)

Finding Functions

- Ghidra has an experimental plugin for exploring how functions already found in a program begin and using that information to find additional functions.
- To enable it from the Code Browser: **File** → **Configure...**, click on the (upper right) plug icon, and select the **Function Bit Patterns Explorer** plugin.
- Then select **Tools** → **Explore Function Bit Patterns** from the Code Browser.
- Hovering over something in the tool and pressing **F1** will bring up the Ghidra help (this works for most parts of Ghidra).

Finding Functions

- The general strategy is to explore the instruction trees and byte sequences, select/combine/mine for interesting patterns, then send them to the **Pattern Clipboard** for evaluation. See the help for details.
- Another useful feature is the **Disassembled View** (accessed through the **Window** menu of the Code Browser). This allows you to see what the bytes at the current address would disassemble to without actually disassembling them.

Contents

Improving Decompilation: Data Types
- Defining Structures
- Defining Classes
- Decompiling Virtual Function Calls

Defining Data Types

- One of the best ways to clean up the decompiled code is to define data structures.
- You can do this manually through the **Data Type Manager**.
- You can also have Ghidra help you by right-clicking on a variable in the decompiler view and selecting
  - **Auto Create (Class) Structure**, or
  - **Auto Fill in (Class) Structure**.
- Note: If you happen to have a C header file, you can parse data types from it by selecting **File** → **Parse C Source...** from the Code Browser (doesn't support C++ header files yet).

Exercise: Auto-creating Structures

1. Open and analyze the file **createStructure**.

   This file contains two functions of interest: **setFirstAndThird** and **setSecondAndFourth**.

   The first parameter to each of these two function has type **exampleStruct ***, where **exampleStruct** is defined as follows:

```
typedef struct {
    long a
    int b
    char *c;
    short d
} exampleStruct;
```

Exercise: Auto-creating Structures

2. Navigate to **setFirstAndThird**.
3. In the decompiler view, change the type of the second parameter to **long** and the third parameter to **char ***
4. In the decompiler view, right-click on **param1** and select **Auto Create Structure**.
5. Right-click on the default structure name (**astruct**) in the decompiler and select **Edit Data Type...**
6. Change the name of the structure to **exampleStruct** and the names of the defined fields to **a** and **c**.
7. Note that this isn't all of the fields in the structure, just the ones that were used in this function.

   (continued)

Exercise: Auto-creating Structures

8. Now navigate to **setSecondAndFourth**.
9. Change the type of the first parameter to **exampleStruct ***, the type of the second to **int**, and the type of the third to **short**.
10. Right-click on the first parameter and select **Auto Fill in Structure**.
11. Edit the structure again to add the names from the structure definition for the new fields (you can also select each field in the decompiler and press **L**).
12. Revel in how much better the decompilation of the two functions looks!

Defining Classes

- If a variable is known to be a **this** parameter, right-clicking on it will yield a menu with the option **Auto Fill in Class Structure** instead of **Auto Fill in Structure**.

Exercise: Defining Classes

1. Open and analyze the file **animals**.
2. In the Listing, press **G** (goto). In the resulting pop-up, enter **getAnimalAge**.
3. This will bring up the **Go To...** dialog, where you can select between the two functions with the name **getAnimalAge** (the functions are in different namespaces).

   Note: There are other windows, such as the **Functions** window, in which there is no default namespace column. You can add a namespace column by right-clicking on any column name and selecting **Add/Remove Columns...** You can also configure the display of certain columns by right-clicking on the column name.

   (continued)

Exercise: Defining Classes

4. Select **Dog::getAnimalAge** in the pop-up. This will cause the Code Browser to navigate to **Dog::getAnimalAge()**.

   Note: Alternatively, you can quickly navigate to the functions in a class using the **Classes** folder of the **Symbol Tree**.

5. Verify that in the decompiler view, right-clicking on the token **Dog** yields a menu with **Auto Fill in Class Structure** as an option. Note that Ghidra has already created an empty structure named **Dog**.

Exercise: Virtual Function Tables

1. Here is what the end of **main** looks like in the source code:

```
Animal *a;
...
a->printInfo();   //non-virtual
a->printSound();  //virtual
a->printSpecificFact(); //virtual
int animalAge = a->getAnimalAge(); //virtual
delete(a);
return animalAge;
```

   Navigate to the function **main** and examine Ghidra's decompilation.

   (continued)

Exercise: Virtual Function Tables

2. The task is to get the names of the virtual functions to show up in the decompiler. At a high level, the steps are:
   - For each virtual function **foo** of the class **Animal**, create a function definition, which is a data type representing the signature of **foo**.
   - Create a data type for the vftable of **Animal**. This data type will be a structure whose fields are the function signature data types (in order).
   - Change the first field of the **Animal** data type to be a pointer to the vftable data type.

   (continued)

Exercise: Virtual Function Tables

3. First, create a function definition for each of the virtual functions

    - **void printSound(void)**
    - **void printSpecificFact(void)**
    - **int getAnimalAge(void)**

    by right-clicking on **animals** in the **Data Type Manager** and selecting **New** → **Function Definition...**

    For each function, enter the signature and select **__thiscall** for the calling convention.

Exercise: Virtual Function Tables

4. Now, right-click on **animals** in the **Data Type Manager** and select **New** $\rightarrow$ **Structure...**
5. Give the new structure the name **Animal_vftable**.
6. Fill in the structure with the data types corresponding to the virtual functions of the class **Animal**. You can do this by double-clicking in an entry in the **DataType** column and entering a name of a virtual function.

   Notes:

   - The order of the functions in the vftable is the same as the order they are called in the source code snippet.
   - Be sure to give each field in the vftable structure a name (use the name of the corresponding virtual function).

(continued)

Exercise: Virtual Function Tables

7. Alternatively:
   - Find the vftable for **Animal** (from the Code Browser, **Search** → **For Address Tables...**) and look for the table consisting of calls to **__cxa_pure_virtual**.
   - Apply the three function definition data types to the pointers in the table in the appropriate order.
   - Select the table in the Listing, right-click, **Data** → **Create Structure**
8. In main, re-type the variable passed to **printInfo** to have type **Animal *** and re-name it to **a**.
9. Right-click on **a** and select **Auto Fill in Structure** (note that this does not say **Auto Create Structure** since Ghidra automatically created a default empty **Animal** structure).

Exercise: Virtual Function Tables

10. Finally, edit the **Animal** structure itself so that the first field is an element of type **Animal_vftable *** with name **Animal_vftable**.
11. Verify that the virtual function names appear in the decompilation of **main**.

Contents

Improving Decompilation: Function Calls

- Introduction
- Function Signatures: Listing vs. Decompiler
- The Decompiler Parameter ID Analyzer
- Overriding a Signature at a Call Site
- Custom Calling Conventions
- Multiple Storage Locations
- Inlining Functions
- System Calls

Function Signatures and Calls

- In this section, we focus on issues involving function signatures and function calls.

Refresher on Function Signatures in Ghidra:

- Sometimes the signature of a function shown in the Listing (or in the **Functions** window) will not match the signature shown in the decompiler.
- This happens because the decompiler performs its own analysis to determine the function's signature.
- The decompiler re-analyzes the function each time it is decompiled.
- The signature shown in the Listing is created when the function is (re-)created. This is the signature that is stored in the Ghidra program database.

Refresher on Function Signatures in Ghidra:

- To transfer the decompiler's signature to the Listing, right-click on the function in the decompiler and select **Commit Params/Return**. The transfered signature will be saved to the program database.
- The situation is the same for the local variables of a function: right-click on the function in the decompiler and select **Commit Locals**.

  Note: Usually it's better not to commit locals and instead to let the decompiler assign types to them automatically. Committing locals can interfere with type propagation.
- Editing a function's signature manually, from either the Listing or the decompiler, commits the new signature to the program database.

Decompiler Parameter ID

- The **Decompiler Parameter ID Analyzer** (**Analysis** → **One Shot** → **Decompiler Parameter ID**) uses the decompiler and an exploration of the call tree to determine parameter, return type, and calling convention information about functions in a program. This analyzer can be quite useful when you have some rich type information, such as known types from library calls. However, if you run this analyzer too early or before fixing problems, you can end up propagating bad information all over the program.
- Note: this analyzer will commit the signature of each function.

Overriding Signatures

- It is possible to override a function's signature at a particular call site.
- This is basically only ever needed for variadic functions (functions which take a variable number of arguments), or to adjust the arguments of indirect calls. In other cases you should edit the signature of the called function directly.
- To override a signature, right-click on the function call in the decompiler and select **Override Signature**.
- To remove an override, right-click and select **Remove Signature Override**.

Aside: The System V AMD64 ABI

For reference when doing the exercises, here is the calling convention used by Linux on x86_64:

- First 6 integer/pointer args are passed in **RDI**, **RSI**, **RDX**, **RCX**, **R8**, **R9**.
- First 8 floating point args are passed in **XMM0-XMM7**.
- Additional args are passed on the stack.
- For variadic functions, the number of floating point args passed in the **XMM** registers is passed in **AL**.

Exercise: Overriding Signatures

1. Open and analyze the file **override.so**, then navigate to the function **overrideSignature**. Override the signature of the call to **printf**, if necessary, using the format string to determine number and types of the parameters to the call. Some of the parameters to **printf** are global variables; determine and apply their types.

Exercise: Overriding Signatures

(advance for solution)

Exercise: Overriding Signatures

(advance for solution)

- Signature:

  printf(char *,int,long,double,char *,int,int,int,int)

- Types:

  a: **int**

  b: **long**

  c: **double**

  d: **char ***

Custom Calling Conventions

- Sometimes a function will use a non-standard calling convention.
- In such a case, you can set the calling convention manually.
- To do this, right-click on the function in the decompiler and select **Edit Function Signature**.
- In the resulting window, select **Use Custom Storage** under **Function Attributes**.

Exercise: Custom Calling Conventions

1. Open and analyze the file **custom**, then navigate to the function **main**.
2. **main** calls the functions **sum** and **diff**, which have custom calling conventions.
3. Examine the bodies and call sites of **sum** and **diff** to determine their signatures and custom calling conventions.
4. Edit each of the two functions and select **Use Custom Storage**.
5. Type the correct signature into the text window and press enter.

   (continued...)

Exercise: Custom Calling Conventions

6. Click on the entries in the **Storage** column to set the storage for each parameter/return value.
7. In the resulting **Storage Address Editor** window, click **Add** to add storage, then click on each table entry to modify.
8. You might find it helpful to remove some of the variable references Ghidra adds in the Listing, particularly to stack variables. To do this, **Edit** → **Tool Options** → **Listing Fields** → **Operands Field** from the Code Browser.

Exercise: Custom Calling Conventions

(advance for solutions)

Exercise: Custom Calling Conventions

(advance for solutions)

- **long sum(long, long)**: return in **RAX**, args in **R14, R15**.
- **long diff(long, long)**: return in **RBX**, args in **[RSP + 0x8]**, **[RSP + 0x10]**

Multiple Storage Locations

- You may have noticed that you can add multiple storage locations for one parameter when editing a function signature.
- This is used (for example) for functions which return **register pairs**.

Exercise: Multiple Storage Locations

1. Open and analyze the file **ldiv**, then navigate to the function **main**.
2. In the decompiler, right-click on the call to **ldiv** and select **Edit Function Signature**. How does **ldiv** use multiple storage locations for a function variable?

   (advance for solution)

Exercise: Multiple Storage Locations

- The result of **ldiv** is returned in the register pair **RDX:RAX** (**RAX** contains the quotient, **RDX** contains the remainder).

Inlining Functions

- Some special functions have side effects that the decompiler needs to know about for correct decompilation. You can handle this situation by marking them as **inline**.
- If **foo** is marked as inline, calls to **foo** will be replaced by the body of **foo** during decompilation.
- To mark **foo** as inline, edit **foo**'s signature and check the **In Line** function attribute.

Inlining Functions

- Inlining a function is related to the notion of a **call fixup**, where calls to certain functions are replaced with snippets of Pcode.
- These functions are recognized by name and have the call fixup applied automatically.
- Examples include functions related to structured exception handling in Windows.
- You can also select from pre-defined call fixups when editing a function signature.
- Note: there are no fixups defined for x86_64 binaries compiled with **gcc**, so the **Call Fixup** selector is greyed out for the exercise files.

Exercise: Inlining Functions

1. Open and analyze the file **inline**, then navigate to the function **main**.
2. When provided with the correct number of command line arguments, this function should parse **argv[1]** and **argv[2]** into unsigned long values and print their sum. The task is to get the decompiler to show this.
3. First, ensure that **main** has the correct signature (**int main(int argc, char **argv)**).
4. Next, override the signature of the call to **printf** if necessary, so that it agrees with the format string.

   (continued)

Exercise: Inlining Functions

5. The decompilation will still be incorrect. Marking **adjustStack** and **restoreStack** as inline yields correct decompilation. Why?

Exercise: Inlining Functions

5. The decompilation will still be incorrect. Marking **adjustStack** and **restoreStack** as inline yields correct decompilation. Why?

adjustStack decreases the stack pointer by 16, which violates the calling convention. Since the default behavior of the decompiler is to assume that a function follows the calling convention, it assumes that the call to **adjustStack** does not change the value of the stack pointer. This assumption leads to incorrect analysis. If you mark **adjustStack** and **restoreStack** as inline, their bodies will be incorporated into **main** during decompilation and the changes to the stack pointer will be tracked.

System Calls

- **System calls** are a way for a program to request a service from the operating system.
- Examples include process control, file management, device management,. . .
- A typical implementation uses a special native instruction along with a designated register, which we'll call the **system call register**.
- When the special instruction is executed, the value in the system call register determines which function is called.

Exercise: System Calls

1. Open and analyze the file **write**, then navigate to **main**.

   Note: **main** prints `Hello World!` to the screen using the **write** system call.

Before going further, let's examine what we see.

- In the decompiler, you should see **syscall()**, which looks like a function call but isn't (try clicking on it).
- This is an example of a **user-defined Pcode op**.
- Such operations are used when implementing the Pcode for a particular instruction is too hard (or impossible).
- These operations show up as **CALLOTHER** Pcode ops in the Pcode field in the Listing. They can have inputs and outputs, but otherwise are treated as black boxes by the decompiler.

Exercise: System Calls

2. In the decompiler, why is the return value of **main** `undefined` [16]?

Exercise: System Calls

2. In the decompiler, why is the return value of **main** `undefined [16]`?

- The **SYSCALL** instruction is translated to a single **CALLOTHER** Pcode op (named **syscall**). The decompiler does not consider this operation to have any side effects, so when it tries to automatically determine the return type it sees a move to **RDX** and a move to **RAX** before the **RET** instruction. These registers form a register pair for this architecture, so the decompiler thinks the return value is 16 bytes.
- So how do we improve the decompilation?

Exercise: System Calls

- This system call is a call to **write** since 1 is written to the system call register (**RAX**) before the **syscall** instruction is executed (search online for "x64 Linux syscall table").
- We'd like the call to **write** to appear with the correct name, signature, and calling convention.
- We'd also like cross references, so that we can easily see all calls to **write**.
- During execution, the code for the **write** function is somewhere in the kernel and not in the program's address space.
- So what should the call target be in Ghidra?
- Answer: use **overlay blocks** on the **OTHER** space.

Exercise: System Calls

- Prior to Ghidra 9.1, the **OTHER** space was used to store data from a binary that does not get loaded into memory, such as the `.comment` section of an ELF file.
- In 9.1, we've extended the ability to make references into the **OTHER** space.
- You can't use this space directly, but you can create **overlay blocks** on the **OTHER** space.
- Overlays are a (sort of old school) technique to allow different blocks to be swapped in and out at the same address.
- For our purposes, they allow us to put things in an artificial memory space without the possibility of conflicting with other uses of that space.

Exercise: System Calls

3. Create an overlay of the **OTHER** space as follows:
   (i) Bring up the **Memory Map** by clicking on the ram chip icon in the tool bar of the Code Browser.
   (ii) Click on the green plus to add a block.
   (iii) Call the block **syscall_block**. Have it start at address `0x0` of the **OTHER** space and have length `0x1000`. For Block Type, select **Overlay** from the drop-down menu.

Exercise: System Calls

4. Next, go to address `0x1` in **syscall_block** and create a function (in the Listing, select both the address and the `??` and press `f`).
5. Edit this new function to give it the name **write** and the **syscall** calling convention.
6. If you happen to know the parameters and their types you can add them. Altervatively, select the new function **write** in the Code Browser, right-click on **generic_clib_64** in the **Data Type Manager**, and select **Apply Function Data Types**

   Note: the function we've created has no body. It's essentially an address to hang a function signature and to get cross-references.

Exercise: System Calls

7. Now, navigate back to the **syscall** instruction in **main**.
8. Click on the instruction in the Listing, then press `r` to bring up the **Reference Manager**.
9. Click the green plus to add a reference. Enter **syscall_block::1** for the "To Address" and **CALLOTHER_CALL_OVERRIDE** for the Ref-Type. This reference type essentially transforms the **CALLOTHER** Pcode op to a **CALL** op before sending the Pcode to the decompiler. The call target is the "To Address" of the reference.

   The decompilation should now look as expected.

System Call Notes

1. The script `ResolveX86orX64LinuxSyscallScript.java` will do all of this for you. You can run it on this file, but a better demonstration is to run it on a libc shared object file.
2. The script uses the **Symbolic Propagator** to determine the value of a register at a particular location.
3. The script requires a mapping from system call numbers to system call names. The x86 and x64 ones come with Ghidra, you will need to supply others.
4. Also, the signatures of most Linux system calls are included with Ghidra (used in step 6 above). The script shows you how to apply function data types programmatically, but you might have to supply your own data type archive.

Contents

Improving Decompilation: Control Flow
- Fixing Switch Statements
- Shared Returns
- Control Flow Oddities

Fixing Switch Statements

- Sometimes you will see warnings in the decompiler view stating that there are too many branches to recover a jumptable.
- One reason for this is that there actually is a jump table, but the decompiler can't determine bounds on the switch variable.
- In such cases, you can add the jump targets manually and then run the script **SwitchOverride.java**.
- Note: To find such locations in a program, run the script **FindUnrecoveredSwitchesScript.java**.

Exercise: Fixing Switch Statements

1. Open and analyze the file **switch**, then navigate to the function **main**. The decompiler view should contain a warning about an unrecovered jumptable.
2. The global variable **array** is the jumptable.
3. Navigate to **array** in the Listing and press **p** to define the first element to be a pointer. Note: this will clear any data type information that Ghidra assigned to **array** automatically.
4. Now press **[** to define an array. Enter 10 for the number of elements.
5. This will trigger disassembly at each of the addresses in the jumptable, but these addresses are not yet part of the function **main**.

   (continued...)

Exercise: Fixing Switch Statements

6. Navigate to the **JMP** instruction which jumps to **array** + an offset.
7. Press **R** to bring up the References Editor and click on the mnemonic (**JMP**).
8. You can use the green plus to add a **COMPUTED_JUMP** reference to each address stored in the jumptable one at a time.
9. Alternatively:
    - Select the **JMP** instruction
    - **Select** → **Forward Refs** from the Code Browser.
    - **Select** → **Forward Refs** again.
    - Drag the selection onto the References Editor Dialog.

Exercise: Fixing Switch Statements

10. Right click on the label **main** in the Listing, then select **Function** → **Re-create Function**.
11. The jump targets are now part of **main**, which you can verify by examining the Function graph.
12. Finally, navigate back to the **JMP** instruction and use the Script Manager to run **SwitchOverride.java**.

Shared Returns

- If a **callerOne** ends with call to **callee**, compilers will sometimes perform an optimization which replaces that final call with a jump.
- If **callerOne** and **callerTwo** both end with calls to **callee**, this optimization will result in **callerOne** and **callerTwo** ending with jumps to **callee**.
- The **Shared Return Analyzer** detects this situation and modifies the flow of the jump instruction to have type **CALL_RETURN**. This will change how the functions are displayed in the decompiler.
- You can also do this manually, in case the analyzer missed something (for example, if only one of the functions sharing a final call/jump has been found and disassembled).

Exercise: Shared Returns

1. Uncheck the **Shared Return Calls** analyzer before analyzing **sharedReturn**.
2. This file has been stripped of symbols. To find **main**, navigate to **entry** and look for the call to **__libc_start_main**. The first argument to this call corresponds to the **main** method in the source code.
3. **main** contains two calls to non-library functions. Each callee contains a **JMP** instruction corresponding to what was a function call in the source code.
4. Find these **JMP** instructions, right-click, select **Modify Instruction Flow...**, and change the flow to **CALL_RETURN**. Verify that a new function call appears in the decompilation.

Opaque Predicates

- One anti-disassembly technique is to create an if-else statement with a condition that always evalutes to the same value, but complicated enough for this to be difficult to determine statically.
- This is an example of an **opaque predicate**.
- The branch that is never taken can contain bytes sequences intended to thwart static analysis, such as sequences which disassemble to jumps to invalid targets.

Exercise: Opaque Predicates

1. Open and analyze the file **opaque**, then navigate to the function **main**.
2. **main** contains an opaque predicate. Find it and fix it by either:
   (i) Changing a conditional jump to an unconditional jump using the instruction patcher. To patch an instruction, right-click on it in the Listing and select **Patch Instruction**.
   (ii) Adding a (primary) reference with Ref-Type **JUMP_OVERRIDE_UNCONDITIONAL** on the appropriate conditional jump. The "To Address" of the reference should be the jump target. To the decompiler, this will change the conditional jump to an unconditional jump.

(hint on next slide)

Exercise: Opaque Predicates

Hint: The opaque predicate is based on the fact that if you square an integer and reduce mod 4, you can only ever get 0 or 1. Look for a multiplication, modular reduction (optimized to a bitmask), and comparison in the assembly.

Jumps Within Instructions

- The decompiler can repeatedly disassemble the same byte as part of different instructions as it follows flow.
- The listing can't do this: each byte has to be assigned to one instruction.
- One consequence is that the decompilation can be correct even if the listing shows a disassembly error.
- This can happen when encountering certain anti-disassembly techniques.

Exercise: Jumps Within Instructions

1. Open and analyze the file **jumpWithinInstruction**, then navigate to the function **main**.
2. You should see an error in the disassemly but correct decompilation (with a warning). What's going on?

Exercise: Jumps Within Instructions

(advance for solutions)

Exercise: Jumps Within Instructions

(advance for solutions)

- **eb ff** is **JMP .+1**. After this instruction executes, **ff c0** are the bytes of the next instruction to execute. Clear the instruction corresponding to **eb ff** and then disassemble starting at **ff** to reveal the instructions that execute after **JMP .+1**.
- Note: After clearing and disassembling, right-click on the **SUB** instruction and select **Fallthrough** → **Auto Override**, which will set the fallthrough address to be the address of the next instruction after **SUB** (skipping data). You should verify that setting this override makes the function graph look better.

Contents

Improving Decompilation: Data Mutability

- Changing Data Mutability
- Constant Data
- Volatile Data

Data Mutability

- **Data Mutability** refers to the assumptions Ghidra makes regarding whether a particular data element can change.
- There are three data mutability settings:
  1. normal
  2. constant
  3. volatile
- There are two ways to change data mutability:
  1. Right-click on the (defined) data in the Listing and select **Settings...**
  2. Set the mutability of an entire block of memory through the Memory Map (**Window** → **Memory Map** from the Code Browser).

Constant Data

- The decompiler will display the contents of a memory location if the contents are marked as constant.
- Otherwise it will display a pointer to the location.

Exercise: Constant Data

1. Open and analyze the file **dataMutability**, then navigate to the function **main**.
2. Change the settings of the target of the pointer variable **writeable** to constant by right-clicking and selecting **Data** → **Settings...** in the Listing. Verify that the changes are reflected in the decompiler.
3. Restore the data mutability and change it again by modifying the permissions of the appropriate block in the Memory Map.

Volatile Data

- Marking a data element as volatile tells the decompile to assume that the value of a variable could change at any time.
- This can prevent certain simplifications.

Exercise: Volatile Data

1. Note that the decompiler prints warning comments at the top of **main** indicating that unreachable code blocks have been removed.
2. You can prevent this by selecting **Edit** → **Tool Options** → **Decompiler** → **Analysis** and unchecking **Eliminate unreachable code**.
3. After doing this, you will see the global variable **status** appear in the decompilation. Note that it is set to zero and then tested. This is a hint that **status** might be volatile.

Exercise: Volatile Data

4. Mark the data element labelled **status** as volatile and verify that additional code appears in the decompilation of the function **main** (make sure to re-enable unreachable code elimination in the decompiler if you've disabled it).
5. Note: You might have to override the signature on the call to **printf** to get all of its arguments to appear in the decompilation.

Contents

Improving Decompilation: Setting Register Values

Setting Register Values

- Setting a context register (for example, to select ARM or Thumb mode) is a common reason to set register values in Ghidra.
- Additionally, if you set a register value at the beginning of a function, the value will be sent to the decompiler.
- To set a register value, right-click on an address in the Listing and select **Set Register Values...**
- This can be helpful if a register is used to store a global variable. Additionally, it can sometimes be helpful to set register values when trying to understand a function. The decompiler will perform additional transformations, which may yield a simplified view of how the function behaves in restricted cases.

Exercise: Global Variables

1. Open and analyze the file **globalRegVars.so**, then navigate to the function **initRegisterPointerVar**.
2. This function stores the address of a global variable into a register. Determine the address and the register.
3. Set the value of the register to be the address at the beginning of the functions **setRegisterPointerVar** and **getRegisterPointerVar**. If you do it correctly, **getRegisterPointerVar** should decompile to

```
{
    return c;
}
```

Exercise: Simplifying Transformations

1. Open and analyze the file **setRegister**, then navigate to the function **switchFunc**. Set the switch variable (in **RDI**) to a few different values and observe the effect on the decompiled code.

Contents

Troubleshooting Decompilation
- Identifying Problems in the Decompiled Code
- Potential Causes
- Potential Fixes
- Compiler vs. Decompiler

in_, unaff_, and extraout_

- Occasionally, you may see variables in the decompiler view whose names begin with **in_**, **unaff_**, or **extraout_**.
- **in_** or **unaff_**: this typically indicates that a register is read before it is written (and it does not contain a parameter passed to the function).
- Variables that begin with **extraout_** can occur when the decompiler thinks that a value is being used that should have been killed by a call.

Pcode in the Decompiler View

- Occasionally, you might see Pcode operations in the decompiler code.
- Examples: **ZEXT, SEXT, SUB, CONCAT,...**
- See the "Decompiler" section in the Ghidra help.

Potential Causes

1. The decompiler has a function signature wrong (either the signature of the function being decompiled or one of its callees).
2. A common situation is some kind of size mismatch, for example, the decompiler thinks that a call returns a 32-bit value but sees all of **RAX** being used. But then where did the high 32 bits come from?
3. There's a register that actually contains a global parameter or is set as the side effect of a called function.

Potential Fixes

- To fix these issues, the first step is to try to determine if the decompiler is making an assumption that's false.
- Oftentimes, you can correct such errors by:
  - correcting function signatures
  - correcting sizes of data types
  - marking functions as inline
- For example, if you see **in_RAX** in the decompiled view, you should check if there's a call to a function whose return type is mistakenly marked as **void**.

Useful Tools

- Script: **FindPotentialDecompilerProblems.java**: Decompiles all functions in a program, looks for problems, and displays them in a navigable table.
- Script: **CompareFunctionSizesScript.java**: Decompiles all functions in a program and displays a table which contains the size of each function (in instructions) and the size of each decompiled function (in Pcode operations). If a function has many instructions but the decompiled version is small, there could be an incorrect assumption regarding the return value.
- From the Code Browser, **Edit** → **Tool Options...** → **Decompiler** → **Analysis** → uncheck **Eliminate unreachable code**: might help diagnose issues.

Compiler vs. Decompiler

- Sometimes compilers can prove certain facts about special cases and use these facts to emit optimized code.
- This can have consequences for the decompiled code.
- This isn't an error, just something to keep in mind.

Exercise

1. Open and analyze the file **compilerVsDecompiler**.
2. The functions **calls_memcmp** and **calls_memcmp_fixed_len** implement **memcmp** using the **CMPSB.REPE** instruction.
3. Compare the decompiled view of these two functions. What differences do you see?
4. What accounts for these differences? (hint: examine the assembly code)
5. Note: To compare two functions side-by-side, bring up the **Functions** window (**Window** → **Functions** from the Code Browser), select the two functions, right click and select **Compare Functions**. Use the tabs to switch between the Listing and Decompiler views.

Solution

(advance for solutions)

Solution

(advance for solutions)

1. **calls_memcmp_fixed_len** contains **in_ZF** and **in_CF** in the decompiled code, whereas **calls_memcmp** does not.
2. In **calls_memcmp_fixed_len**, the compiler knows that the loop will be executed at least once (**RCX** is set to 8).
3. However, in **calls_memcmp**, the loop might be executed 0 times (**RCX** is set to **param3**).
4. This means that the compiler must initialize the flags **ZF** and **CF** in **calls_memcmp**, but does not have to in **calls_memcmp_fixed_len**, since the loop is guaranteed to execute at least once and that comparison will set the flags.

(continued)

Solutions

6. This is the purpose of the **CMP RDX,RDX** instruction **calls_memcmp** (which does not occur in **calls_memcmp_fixed_len**).
7. The decompiler doesn't do the analysis to prove that a loop must execute at least once.
8. So in the decompiler's view, the values in **ZF** and **CF** at the beginning of **calls_memcmp_fixed_len** might contribute to the return value (in the "case" when the loop body does not execute).