The CodeReason Framework

Trail of Bits

1. Introduction

This is the CodeReason framework design document, tool user manual, API user guide, and milestone delivery guide. This document should serve as the primary source of information about all things CodeReason.

2. Building the framework

2.1 Building on Linux or Mac OS X

CodeReason's install.sh script will install dependencies, install LLVM, compile libVEX, and then perform a build of CodeReason on Ubuntu 12.04. Default default options and paths are used. The resulting tools will be in build/bin/. To use the installation script, run the following commands:

```
chmod +x install.sh
sudo ./install.sh
```

To build on Linux or OS X manually, first build the accompanying libVEX source distribution. To do this, open a shell in the libVEX directory and type 'make'.

Then, from the root of the CodeReason base directory, create a 'build' directory, cd into this directory, and run

```
cmake ..
```

Note that it is important for your PATH to contain `llvm-config`, which was installed earlier by installing LLVM. `llvm-config` is invoked by the CMake build system to determine the paths to LLVM libraries and include files and automatically populate them.

2.2 Building on Windows

Building on Windows also uses CMake. LLVM does not build `llvm-config` for Windows, so the location of the LLVM installation is hardcoded to c:\LLVM . When compiling LLVM, compile LLVM in either debug or release mode, depending on what mode you compile CodeReason in (the modes must match) and install to c:\LLVM.

Building on Windows is supported either by a Visual Studio solution file or an NMake Makefile.

2.2 Building manually

The framework is configured and built using CMake. The framework should build on Linux, OSX, and Windows. The framework links against libVEX from Valgrind, which is shipped with the distribution and can be compiled on Linux and OSX. Compiling libVEX for Windows is more complicated, so a distribution of libVEX has been provided which is pre-compiled.

The framework has a few dependencies:

- LLVM 3.0 or LLVM 3.2

- Boost
- CMake
- Google Protocol Buffers

To build CodeReason manually or on other distributions, LLVM needs to be built only with ARM and X86 support. To build and install LLVM, it's recommended to use CMake. Unpack the LLVM source distribution to a directory on the building computer, open a shell and change directory to the root of the unpacked distribution, and run:

```
    cmake .. -DLLVM_REQUIRES_RTTI=1 –
DLLVM_TARGETS_TO_BUILD=X86;ARM –DCMAKE_INSTALL_PREFIX=”/path”
```

Once CMake is finished, it will have produced a building infrastructure for your platform. On Linux and OS X, by default these are makefiles. On Windows, the default is a Visual Studio solution, it is recommended to change this to NMake makefiles by adding *–G "NMake Makefiles"* to the command line to CMake.

Once LLVM has compiled, use 'make install' to install it into the location specified.

LLVM changed the interface to their instruction decoding API from version 3.0 to 3.2. CodeReason should compile using either LLVM 3.0 or 3.2, and on some newer platforms, LLVM 3.0 will not compile. If LLVM 3.0 will not compile, it's probably okay to use LLVM 3.2 instead.

CodeReason was built and tested using protoc/protobuf 2.4.1, but should be compatible with any later version of protoc/protobuf. Both protoc and protobuf must be installed to build CodeReason. On Ubuntu and Debian, the following packages provide enough of the environment to build CodeReason:

build-essential
g++
cmake
libboost-dev
libboost-thread-dev
libboost-system-dev
libboost-filesystem-dev
libboost-program-options-dev
libboost-date-time-dev
libboost-regex-dev
libprotobuf-dev
libprotobuf-lite7
libprotobuf7
libprotoc7
protobuf-compiler

3. Tools provided

The framework is a lot of library code with some tools that expose different parts of the frontend. The tools do not represent every possible use of the framework, just the ones that have been implemented so far.

3.1 RopTool

RopTool is a tool that will search through supplied containers of executable code to find code that matches certain criteria. These criteria are specified as lua scripts that define preconditions and post conditions for basic blocks in an executable module.

3.1.1 Using RopTool

To use, you supply a script that defines preconditions and post conditions using the lua API. This API is described in more detail in section 6 of this manual. You also supply a path to a module to search, and on the command line you supply the architecture of the code to search for and the input file format.

RopTool is non-interactive. Once launched, it will search for blocks that match the conditions described using the VEE. When all inputs have been exhausted, RopTool will exit and print out the contents of the blocks it found that match the conditions.

3.1.2 Command-line options

Command-line options for RopTool are given by running the program with –*help*. Generally, the following options must be supplied:

- One of *--pe*, *--mach* to indicate the input file format
- *-f* given a path to a PE file to scan
- *-i* given a path to a lua script that defines the pre and post conditions for the search
- *-a* and the architecture of the code to search, usually *X86*

3.2 VEEShell

VEEShell contains much of the functionality that RopTool does, however VEEShell is a command-line interactive tool that primarily helps in exploring the IR, devising queries, and debugging the framework.

3.2.1 Using VEEShell

To use, the user supplies command line parameters indicating the file format and architecture. After parsing and processing the input file, the user is placed into a shell where they can view and explore the IR.

3.2.2 Command-line options

Command-line options for VEEShell are …

3.3 ImgTool

ImgTool parses executable containers and prints out the sections that would be identified by CodeReason components that are searching for

3.3.1 Using ImgTool

To use, the user supplies …

3.3.2 Command-line options

The command line options for ImgTool are …

3.4 RopTool2

RopTool2 is a demonstration of using the C++ API to select blocks based on their static semantics.

3.4.1 Using RopTool2

RopTool2 is of limited use, it will only search for blocks that end in a return and add 2 integers together. It is primarily a demonstration of the capabilities of the C++ API to the IR.

3.4.2 Command-line options

To use RopTool2 …

4. Intermediate Representation

The CodeReason IR is the IR that Valgrind tools have for dynamic analysis and instrumentation of applications.

5. C++ API

The C++ API is separated into a few logical components. There is a component that disassembles bytes of native code and representing them as blocks of IR. There is a component that represents blocks of IR in a class hierarchy and allows for querying of properties. There is another component that represents the concolic evaluation of IR in an environment where registers and memory may have certain defined values.

5.1 The BasicIR C++ API

This IR has a class hierarchy that represents statements in the CodeReason IR.

5.2 The VEE C++ API

The VEE library contains infrastructure to evaluate the result of statements in the BasicIR.

5.3 The RopLib library API

You use the RopLib API by instantiating a RopLibSearcher class. This class is constructed with the following signature:

RopLibSearcher(RopLibVisitorPtr v, std::string sourceFile, std::string f, FileFormat fmt, TargetArch t);

You supply a Visitor, and source files, target files, the file format, and the target architecture. The Searcher class makes itself ready to recover blocks by using the **getBlocks** method.

Once the Searcher class has been populated with blocks, you can query it for the number of blocks that you can search. Searching one block is possible using the **evalOneBlock** method. For each call to **evalOneBlock**, your Visitors callback routine will be invoked once.

The RopLibSearcher class has been specialized into a class that performs a graph-theoretic traversal with state. This class, StatefulRopLibSearcher, has a constructor with identical signature to its superclass and only overrides the **evalOneBlock** method. The primary difference between these two Searchers is the way they use your supplied Visitor class.

The RopLibVisitor class has three abstract methods. One of them, **keepBlock**, is used by the RopLibSearcher class, and the other two, **initialState** and **exploreBlock** are used by the StatefulRopLibSearcher class. Visitor implementations must define method bodies for all three functions but only need to provide functionality for the Searcher they plan to use.

Now we'll discuss the three callbacks that the RopLibVisitor class exposes.

VisitorResult keepBlock(BlockPtr b)

This callback is used by the default RopLibSearcher. Its functionality is fairly straightforward. The Searcher will invoke it for every block in the program that the Searcher has identified and ask if the Visitor wants to keep or discard the block.

The Visitor can use any mechanism to determine whether or not to keep the block.

CodeExplorationStatePtr initialState(void)

This callback is used to initiate a stateful query of the program. The Visitor defines a class that inherits from CodeExplorationState and instantiates a new instance of it, either in its constructor or in this method. It returns a unique pointer to that instance from this function to begin the stateful query of the graph. This represents what the Visitor believes is the initial state of a gadget sequence.

CodeExplorationResult exploreBlock(BlockPtr b, CodeExplorationStatePtr p)

This callback is invoked for every block that the stateful searcher encounters. It gives a pointer to current state and the current block being examined. The CodeExplorationResult class defines a set of virtual addresses that represent the addresses of blocks, which could potentially follow from this block. Adding a VA to that set will cause the stateful traverser to traverse to those blocks. The CodeExplorationResult class also expects to receive a new CodeExplorationState pointer. This callback should instantiate a new CodeExplorationState object and assign it to the CodeExplorationResult class for return.

CodeExplorationState

This class can contain any arbitrary value or collection of values that is useful to the visitor. The stateful Searcher will only use the CodeExplorationState pointer that is valid or meaningful for a particular block, so the visitor can use this to maintain state between queries across blocks.

By combining these callbacks and the appropriate Searchers, complicated queries can be made of program logics. Some examples are included in tools/RopTool and tools/RopScore.

6. LUA API

The lua API is used to specify pre and post conditions. Lua scripts that RopTool consumes are evaluated in their toplevel environment. A '*vee*' object is present which contains a method to register pre and post condition callbacks. Typically, the toplevel script logic will at least register its callbacks, though it may do anything else that a lua program would do.

6.1 lua VEE API

The VEE API allows the user to specify pre and post conditions to the emulation environment. The VEE API takes the form of methods to set the virtual state, and to query the virtual state. If a value in the virtual state is undefined or invalid, the lua type for nil is returned to the script.

6.2 Examples of the lua API

These examples come from the *scripts* directory in the CodeReason source distribution.

TODO: add specific examples

7. Criteria language

The criteria language is a domain specific language that represents conditions that must hold about a block. The criteria language was created after discussion amongst principles and experts at Trail of Bits.

The criteria language is strongly influenced by the syntax of the transfer statement output. The language is a sequence of definitions. Each definition either defines a register write, a memory write, or a temporary value that can be used to bound the conditions around a register or memory write.

The parser, internally, converts the text-representation of exit criteria and produces *Statement* objects. These are the same objects used by the IR, so comparison between statements is possible from the frameworks perspective.

7.1 Criteria language specification

The language is specified as a collection of statements that define an environment when a block is acceptable. Each statement in the criteria language takes the form of

<location> = <expression>

where *<location>* specifies a register, memory location, or temporary value and *<expression>* is any computation or logical operation supported by the language.

The definition for both sides is provided in a recursive grammar written in Boost::Spirit. A benefit to this is that any combinations of nested expressions are legal to the parser and the semantics.

7.1.1 Value read syntax

To indicate that a register or value is read, it should be left on its own. For example, the following statement:

EAX = EBX

This statement says that at the end of the block, the register EAX should contain the value read from the register EBX. More complicated statements are possible, such as:

EAX = MEMREAD[EBX]

Which indicates that EAX should contain the value stored in memory indexed by the contents of the register EBX.

Wildcard criteria are possible with registers by using the '?' identifier. For example:

? = MEMREAD[EBX]

This statement indicates that any register should hold the value returned from reading memory indexed by EBX. Additionally, statements such as the following are possible:

? = MEMREAD[?]

Which says that any register should hold the value returned from reading memory indexed by any register.

Constant integers can also be supplied,

7.2 Criteria language examples

Examples are contained in the *scripts/lang* directory. Examples demonstrate:

- Register equivalence (*reg_equiv*)
- Memory location equivalence (*mem_equiv*)
- Register and memory equivalence with epsilon (*reg_equiv_eps* and *mem_equiv_eps*)
- Wildcard register assignment from memory (*wild_reg_assign*)
- Wildcard register binary operations (*wild_binary_ops*)

8. Milestones

Each milestone delivery is documented in this section.

8.1 Milestone 1

8.1.1 Task 1

There are unit tests within tools/test/test.cpp that test the abstract register functionality, the test is contained within Script.ScriptAbstrReg. By running *codeReasonTest*, you can test the abstract register subsystem.

8.1.2 Task 2

The *scripts/alu* directory contains a collection of lua scripts that find instruction sequences useful in the construction of an ALU. These scripts demonstrate RopTools efficacy at finding sequences.

To execute each test, run RopTool as follows:

```
./bin/RopTool --pe -f ../tests/sequence_holder.dll -i ../scripts/valu/<OP>.lua
```

For each test run, replace <OP> with one of mul, add, sub, xor, or, and, shl, shr, or div.

RopTool will work for a while, and then output the blocks that it found. You should be able to find sequences for each file that perfrom the needed operations.

You will need to execute RopTool once for each lua file within the *valu* directory. For each invocation, examine the output and note that a block has been found with each following address

Sequence locations within sequence_holder.dll are in the following table

mul	10010fa4, 10010fbc
add	10010fbe, 10001090
sub	100010a0, 10006f6a
xor	100010f0
or	100010e0, 10007b50
and	100010d0, 100105ac
shl	10001080
shr	10001070
div	100010c0

8.2 Milestone 2

8.2.1 Task 2

This was broken down into 2 subtasks that can be tested basically the same way.

These changes can be demonstrated both using VEEShell and also by using the newly created RopTool2.

Once the tool has been built, run the following command from the build directory:

```
$ ./bin/VEEShell -a X86 -f ../tests/testSub.bin
```

You should get the following output:

```
blockLen: 6
>
```

From here, you have shell-like access to the IR and the concolic execution engine. To demonstrate codeReasons understanding of the semantics of the given sequence, press 'r' and hit enter:

```
> r
OP = I:U32(0x6)
DEP1 = REGREAD(ECX)
DEP2 = I:U32(0x1)
NDEP = I:U32(0x0)
EAX = Sub32[ REGREAD(ECX), I:U32(0x1) ]
ESP = Add32[ REGREAD(ESP), I:U32(0x4) ]
EIP = REGREAD(ESP)
```

This shows us how the outputs of the block are affected by the statements in the block. This information is retrieved by helper code prepared for this milestone, located in libs/IR/Helpers.cpp.

With this, we can now move on to consider RopTool2. RopTool2 is not user-scriptable yet, that feature will be added as part of task 3. RopTool2 uses the transfer semantics of the IR to identify sequences that have the result of adding two distinct GPRs together. Its code can be found in tools/RopTool2/RopTool2.cpp and the logic that uses the transfer semantics is stored in HasAddAndReturn::keepBlock.

To see RopTool2 extract results, you should be able to do the following:

```
$ ./bin/RopTool2 --pe -f ../tests/b.dll -a X86
building blocks...
searching ...
Done |  Elapsed | Remaining |     Processed |   Unprocessed | Rate
done searching  |  00:00:01 | 20,131 blocks |  3,493 blocks | 2,012 blocks/s
found 8
10007268
1000726b
1000726d
10007271
10007273
10007274
10007276
1000727e
```

From IDA, we can verify that the instructions contained at 1000727e satisfy the criteria for our search:

```
.text:1000727E                   add     edx, ebx
.text:10007280                   pop     ebx
.text:10007281                   retn    10h
```

With this infrastructure in place, we can begin to develop a language to search for sequences containing specific and well defined properties

8.3 Milestone 3

Milestone 3 demonstrates the design and implementation of the block output specification language. The design of the specification language and examples are above in section 7. The demo scripts included can be parsed with the TestParser shipped with this release of CodeReason. The TestParser will read the contents of a file and then print the CodeReason IR statements that result from reading the file.

The progress made here is that the framework can accept as input text specifications of block transfer semantics. It can then take this text specification and represent it internally in the same IR and data structures that the transfer semantics are represented in.

Next, we can take these semantics and apply search techniques to identify which blocks match criteria without needing to perform concrete evaluation of the IR.

Demo files contained in scripts can be parsed using TestParser as follows:

- *mem_equiv*
    - *./bin/TestParse -i ../scripts/lang/mem_equiv* should output
        - *MEMWRITE(REGREAD(ECX)) = MEMREAD(REGREAD(EBX))*
- *mem_equiv_eps*
    - *./bin/TestParse -i ../scripts/lang/mem_equiv_eps* should output
        - *MEMWRITE(REGREAD(EAX)) = MEMREAD(Add[REGREAD(EBX) RANGE[I:U32(0x0), I:U32(0xa)]])*
- *reg_equiv*
    - *./bin/TestParse -i ../scripts/lang/reg_equiv* should output
        - *RWRITE(EAX) = REGREAD(EBX)*
        - *RWRITE(ECX) = REGREAD(EDX)*
- *reg_equiv_eps*
    - *./bin/TestParse -i ../scripts/lang/reg_equiv_eps* should output
        - *RWRITE(EAX) = Add[REGREAD(EBX) RANGE[I:U32(0x0), I:U32(0x30)]]*
- *wild_binary_ops*
    - *./bin/TestParse -i ../scripts/lang/wild_binary_ops* should output
        - *RWRITE(ANY32) = Add[REGREAD(EBX) I:U32(UNKNOWN)]*
- *wild_reg_assign*
    - *./bin/TestParse -i ../scripts/lang/wild_reg_assign* should output
        - *RWRITE(ANY32) = MEMREAD(I:U32(UNKNOWN))*

8.4 Milestone 4

This milestone called for the generation of sequence characterizations for existing exploits. A paper, titled "ROP Gadgets in Metasploit", accompanies this document and details our findings. Additionally, the specifications for the gadgets described in that paper accompany this release of CodeReason.

8.5 Milestone 5

This milestone sees the addition of a data serialization and deserialization layer to the IR library. This serialization and deserialization layer allows for code representation to be created once, then stored and consumed multiple times by other tools.

The act of converting program code into the semantics-carrying intermediate representation is computationally expensive. However, de-serializing the stored representation is much less expensive.

To store the intermediate representation, we created a Google Protocol Buffers specification that describes the intermediate representation. This specification is stored in *libs/IR/IR.proto*. An advantage that Protocol Buffers gives is portability. From a given proto definition, the Google protoc compiler can emit adapters for C++, Python and Java to parse serialized code objects.

This allows for other analyses or visualizations to be written in other languages, and consume the output of the IR conversion tool in the Code Reason framework. For example, a Java program using the Processing visualization framework could read a block of serialized protocol buffers information and achieve the same level of understanding about a low-level programs function as the Code Reason framework itself, which is written in C++.

The Protocol Buffers format is also very flexible. In this milestone, we serialize the message describing a module, as a collection of basic blocks, into a file. However, the message can also be serialized across the network, through IPC via a pipe, or stored in a relational database.

Additionally, this serialization allows for the gadget search process to be more efficiently parallelized. Before, individual agents that would process modules of executable code would each have to agree on virtual address boundaries within some source module. Each agent would take a 'slice' of the module as defined by addresses. Then, each agent would decode the instructions within the module and perform the search on those instructions.

With this data serialization layer, one central source can decode all of the instructions in a given module into a series of protobuf messages. Then, the central source can divide up the messages amongst any number of agents and distribute the messages to the agents. The agents ingest the messages as IR and then begin the search directly.

As a demonstration of this capability, we have two new tools in the framework: BlockExtract and BlockReader. A run of BlockExtract should appear as follows:

```
$ ./bin/BlockExtract --pe -a X86 -i ../tests/b.dll --db-out b_dll.db
section   0 of   1
99.99609[======================================== ]
all sections read
23624
Wrote out to file b_dll.db
```

Then, BlockReader can be used to read the block in and pretty-print the resulting IR:

```
$ ./bin/BlockReader -i b_dll.db > reader_out
```

And the reader_out file should contain the IR for the block, for example:

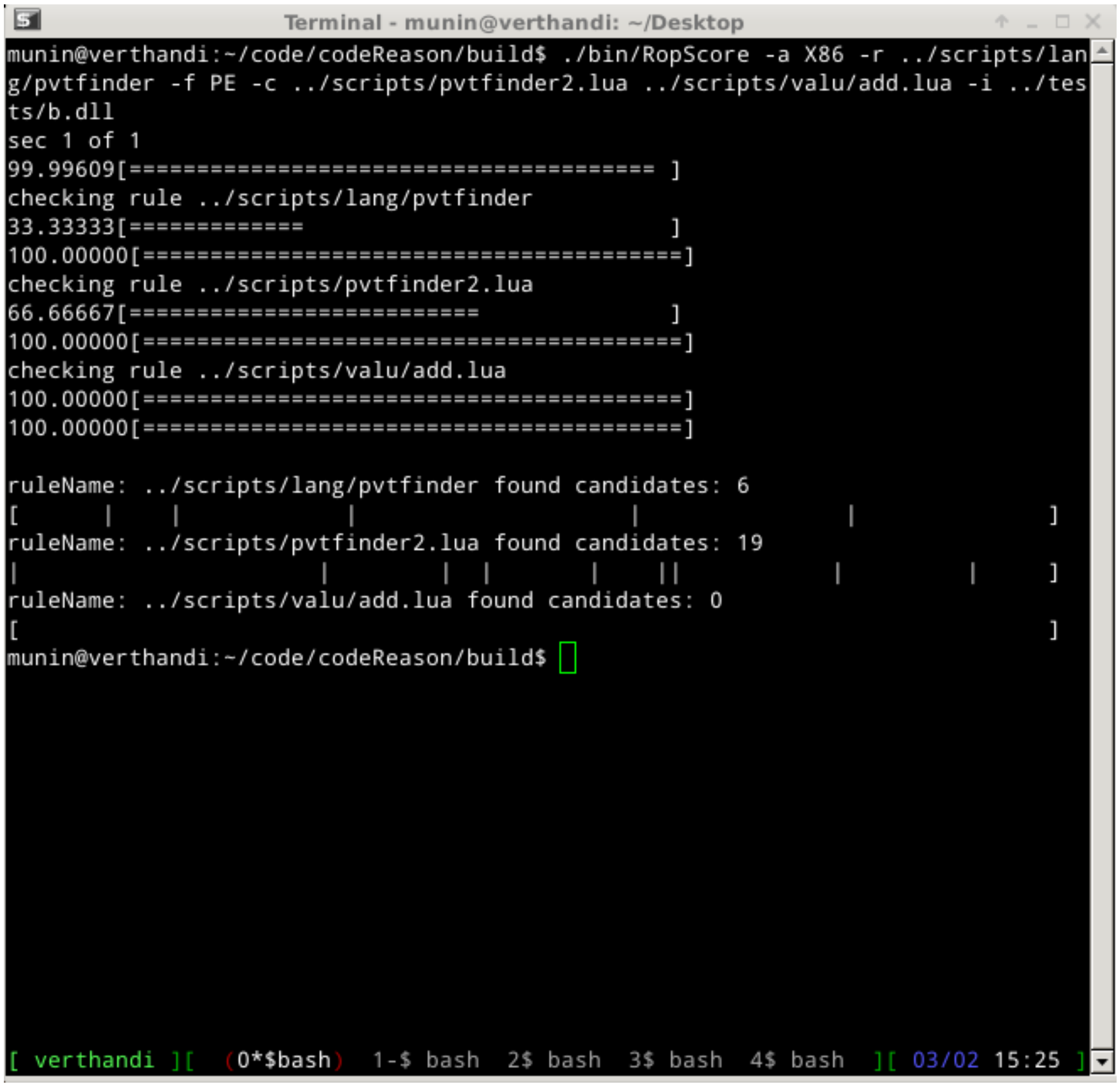
```
$ cat reader_out
...

BLOCK 0x10001009 BLOCK ID: 0
TVALS:
t0:I32
t1:I32
t2:I32
INSTRUCTION DATA:
t0:I32 = REGREAD(ESP)
t1:I32 = MEMREAD(t0:I32)
t2:I32 = Add[ t0:I32 I:U32(0x4) ]
RWRITE(ESP) = t2:I32
RET t1:I32
BLOCK END 0x1000100a

...
$
```

8.6 Milestone 6

This milestone introduces RopScore, a tool that combines elements of the framework developed so far to assign scores to individual modules of how much code can be re-used.

The RopScore tool combines both the concolic-execution engine and the symbolic analysis system. On the command line to RopScore, users can provide any number of both symbolic representations of block effects, and pre / post condition pairs as expressed in LUA scripts. RopScore combines these into a single searcher vector and applies each search tactic to the target program.

The output gives the user a sense of how much code in the program can be re-used, and how distributed or saturated the code identified for re-use is. This gives developers and security professionals an eyeball view of how vulnerable particular modules are.



```
munin@verthandi:~/code/codeReason/build$ ./bin/RopScore -a X86 -r ../scripts/lan
g/pvtfinder -f PE -c ../scripts/pvtfinder2.lua ../scripts/valu/add.lua -i ../tes
ts/b.dll
sec 1 of 1
99.99609[======================================= ]
checking rule ../scripts/lang/pvtfinder
33.33333[=============                           ]
100.00000[========================================]
checking rule ../scripts/pvtfinder2.lua
66.66667[==========================              ]
100.00000[========================================]
checking rule ../scripts/valu/add.lua
100.00000[========================================]
100.00000[========================================]

ruleName: ../scripts/lang/pvtfinder found candidates: 6
[      |    |            |                    |              |             ]
ruleName: ../scripts/pvtfinder2.lua found candidates: 19
|                      |        |  |       |    ||           |         |     ]
ruleName: ../scripts/valu/add.lua found candidates: 0
[                                                                          ]
munin@verthandi:~/code/codeReason/build$ 
```

```
[ verthandi ][  (0*$bash)  1-$ bash  2$ bash  3$ bash  4$ bash  ][ 03/02 15:25 ]
```

8.7 Milestone 7

This milestone is more theoretical than the previous 6. In this milestone, we created a framework, algorithm and theory for reasoning about and identifying multi-sequence gadgets. The framework is generic and can be parameterized around different query domains of executable code.

This framework is different from the previous implements, both with the rule-based matching system, which is purely symbolic, and the concolic emulation system. Both of those systems worked on a single block boundary. It was possible for those systems to jump from one block to the next, but maintaining state information across multiple possible blocks is difficult.

Our current rules identify gadgets that are contained within an entire sequence, where all of the matching statements occur in one block. For example, we have created rules that identify a swap of the stack pointer with a general-purpose register followed by a return. However, it could be the case that this sequence could be found in program code across multiple program control boundaries.

Algorithm

Our algorithm to detect multi-sequence gadgets is as follows; parameterized around an abstract state object and an abstract query function that takes as input a block and an abstract state object.

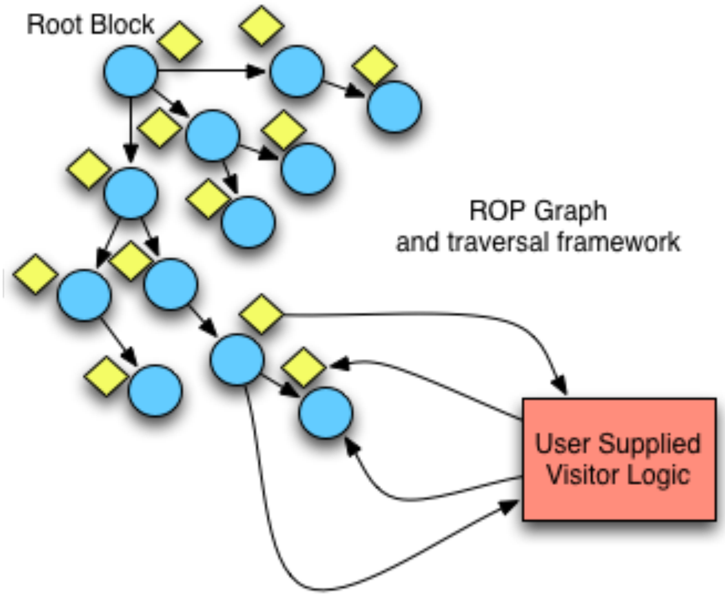
The algorithm will overall build a graph starting from a beginning node. This beginning node is chosen either by a human analyst, or is derived by iterating over every possible node in the program being analyzed. A node in the graph consists of a State object and a Block object.

The algorithm begins by assigning the state object and the initial block object to the root node. The algorithm then invokes the abstract query function with the beginning state and block. The query function returns a set of new blocks to consider, and a new abstract state object. The query function also returns a code to indicate whether or not to continue the search.

If the query function returns a code that indicates the search should continue, and if the query function returns a non-empty set of blocks, the algorithm creates a new node in the graph for every block returned by the query function. To each node, it assigns the appropriate block and the state object returned by the query function. The algorithm populates a stack of unvisited vertices in the graph with all of the vertices just added.

Then, the algorithm visits every vertex in the unvisited vertices stack, applying the same algorithm as above. If a call to the query function ever results in an acceptance, the search is terminated.

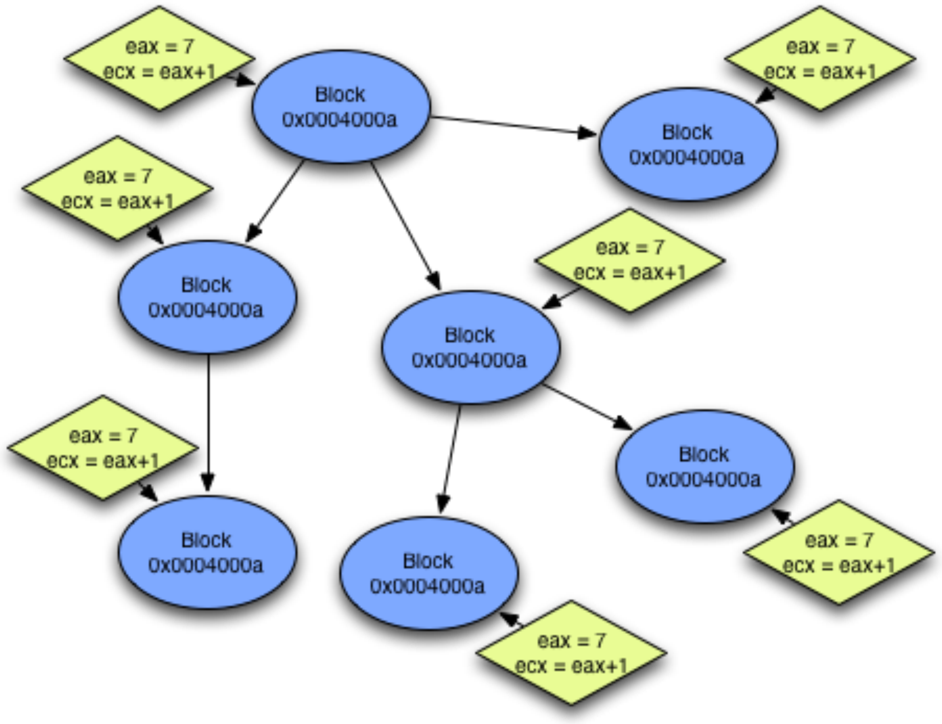
So that we can guarantee termination of this algorithm, we make a compromise where we do not consider gadgets that have loops. This does not mean that ROP programs cannot have loops, just that their gadgets cannot. To enforce termination, we keep track of which nodes we have added to a given walk through the graph and do not add any nodes that we have already visited.



Framework for Symbolic Code Reasoning

The framework breaks down as one class in RopLib that implements the graph search and traversal, and classes within RopTool/RopScore/etc that implement the specific searches. Note that this part hasn't been completed as of this milestone, the completion of those specific searchers is in milestone 8.

A high-level view of the framework appears below. RopTool/RopScore provides the tan components and RopLib provides the blue components. This allows for a few additional applications beyond finding complicated gadget sequences. This framework could be adapted to general purpose binary analysis



The visitor logic provides nodes that follow the block. The visitor logic could identify multiple blocks that could follow; it does not need to identify a specific block that must follow. This allows for the framework to symbolically explore possible futures. Additionally, the framework can record predicates for reachability within the state object and refine as the search continues.

Specific implementation and boundaries

The implementation is contained in libs/RopLib/RopLib.cpp and libs/RopLib/RopLib.h as a series of classes. The documentation for those classes has been added to section 5.3 of this document.

This implementation draws on inspiration from the clang-analyzer project. There, a framework provides a graph structure and abstract state maintenance, while the analysis logic visits each program point and is given the program state at that point.

This framework gives the author of an analysis an easier concept of how to write their queries. Instead of having to worry about tracking program state globally, they just have to concern themselves with querying state at a particular point with a specific expression or block of code.

This concept carries well into the CodeReason framework. An analysis can maintain all of the state information it needs in a state object that is opaque to the state-aware traversal and graph construction code. The analysis is given a block of code and a state and asks, "Do I make progress given the combination of these items". If progress is not made, the analysis can opt to continue exploration. The analysis can also determine that the exploration has proceeded for long enough and to stop the search.

Mass parallelization

An additional advantage that the algorithm has over the previous searching logic within CodeReason is that it is particularly amenable to mass parallelization. Once the algorithm has computed a node and its state value, it is immutable. The exploration of additional paths can be given to other threads or even other processing units on other systems. This allows for the evaluation of programs to be distributed across as many cores as an investigator has at their disposal for a minimal amount of duplicated work.

9. Demos

9.1 Locating pivots in b.dll using concolic execution

This demo uses RopTool. Run the following command:

./bin/RopTool -i ../scripts/pvtfinder2.lua -a X86 -f ../tests/b.dll –pe

RopTool will search for blocks matching criteria and display them, along with their disassembly.

9.2 Locating ROP add statements using transfer semantic evaluation

This demo uses RopTool2. Run the following command:

./bin/RopTool2 -a X86 -f ../tests/b.dll –pe

RopTool2 will search for blocks matching exit criteria only. It will display the virtual address, in hex, of the entry of the block.

9.3 Removing string obfuscation using VEEShell

9.3.1 Motivation

Frequently, malware seeks to obfuscate its operation and intent from analysis. This obfuscation generally takes the form of protecting static constants within the program image. A favorite target of this protection is string literals. Malware will often obfuscate the use of string literals, and many times, the specific mechanism of obfuscation can give some insight into the attackers thought process.

In this example, we'll look at analyzing the string obfuscation functions used in Flame. The samples were obtained from malware.lu. The function is first identified via static analysis in IDA, and is easily identified. There is one function, which precedes other functions that take string literals as arguments, and an argument to the first function is always some non-string global data.

Some manual examination revealed that this string obfuscation system stores global data in a structure, organized something like the following:

```
struct ObfuscatedString {
 char padding[7];
 char hasDeobfuscated;
 short stringLen;
 char string[];
};
```

Each structure has variable-length data at the end, with 7 bytes of data unused. Each structure represents a single protected string.

CodeReason is capable of summarizing the decryption function such that the summarization is the deobfuscator. The code that performs the de-obfuscation is contained within two other functions.

The first function checks the `hasDeobfuscated` field and if it is zero will return a pointer to the first element of the string. If it is not zero, it will call the 2nd function, and then set `hasDeobfuscated` to zero.

The second function will iterate over every character in the `string` array. At each character, it will call the third function and then subtract the value returned by the third function from the character in the string array. So it looks something like:

```
void inplace_buffer_decrypt(unsigned char *buf, int len) {
  int counted = 0;

  while( counted < len ) {
      unsigned char *cur = buf + counted;
      unsigned char newChar = get_decrypt_modifier_f(counted);
      *cur -= newChar;
      ++counted;
  }
  return;
}
```

What about the `get_decrypt_modifier` function? This function is one basic block long and looks like this:

```
lea     ecx, [eax+11h]
add     eax, 0Bh
imul    ecx, eax
mov     edx, ecx
shr     edx, 8
mov     eax, edx
xor     eax, ecx
shr     eax, 10h
xor     eax, edx
xor     eax, ecx
retn
```

The function of this code is not immediately clear. However, Code Reason can examine this block and construct an equation for the output register ‘eax’. This would provide insight into what this block 'returns' to its caller, and would capture the semantics of what get_decrypt_modifier does for use in a deobfuscator.

Getting this from codeReason looks like this:

```
$ ./bin/VEEShell -a X86 -f ../tests/testSkyWipe.bin
blockLen: 28
r

...

EAX = Xor32[ Xor32[ Shr32[ Xor32[ Shr32[ Mul32[ Add32[ REGREAD(EAX),
I:U32(0xb) ], Add32[ REGREAD(EAX), I:U32(0x11) ] ], I:U8(0x8) ], Mul32[
Add32[ REGREAD(EAX), I:U32(0xb) ], Add32[ REGREAD(EAX), I:U32(0x11) ] ]
], I:U8(0x10) ], Shr32[ Mul32[ Add32[ REGREAD(EAX), I:U32(0xb) ],
Add32[ REGREAD(EAX), I:U32(0x11) ] ], I:U8(0x8) ] ], Mul32[ Add32[
REGREAD(EAX), I:U32(0xb) ], Add32[ REGREAD(EAX), I:U32(0x11) ] ] ]

...

EIP = REGREAD(ESP)
```

If functions for Xor32, Mul32, Add32, and Shr32 are implemented, we have this function in C, like so:

```
        unsigned char get_decrypt_modifier_f(unsigned int a) {
          return  Xor32(
                      Xor32(
                          Shr32(
                              Xor32(
                                  Shr32(
                                      Mul32(
                                          Add32( a, 0xb),
                                          Add32( a, 0x11) ),
                                      0x8 ),
                                  Mul32(
                                      Add32( a, 0xb ),
                                      Add32( a, 0x11 ) ) ),
                              0x10 ),
                          Shr32(
                              Mul32(
                                  Add32( a, 0xb ),
                                  Add32( a, 0x11 ) ),
                              0x8 ) ),
                      Mul32(
                          Add32( a, 0xb ),
                          Add32( a, 0x11 ) ) );
        }
```

This could just as easily be written in Python as well, or any other language.

And this decrypts strings stored in the original program.

```
C:\code\tmp>skywiper_string_decrypt.exe
CreateToolhelp32Snapshot
```

This demonstration is a little kludgy, because the code has to be manually identified and extracted into a form that can be consumed by the tool currently. However, a future version of the tool could exist as an IDA plug-in, and this would make the process of describing code to the tool more transparent to the user.

10. FAQ

Q: Does CodeReason represent control flow graphs?

No. This concept isn't known to VEX so CodeReason has no native concept of a CFG.