Project research references:

https://unix.stackexchange.com/questions/229802/convert-executable-back-to-c-source-code - no variable names etc
Furthermore, eth.build is a nice learning platform for Smart Contracts with many examples.
https://ethereum.org/en/developers/docs/ethereum-stack/#ethereum-client-apis ??
Remix is an online environment for developing Smart Contracts. It has some example contracts that can be used as a starting points.
Decompiling : https://suif.stanford.edu/dragonbook/
CProver tools: http://www.cprover.org/cprover-manual/cbmc/tutorial/
Existing decompiler: **https://github.com/MrLuit/evm** ,
https://www.trustlook.com/services/smart.html , https://www.ethervm.io/decompile
https://github.com/crytic/rattle evm static analysis of op codes

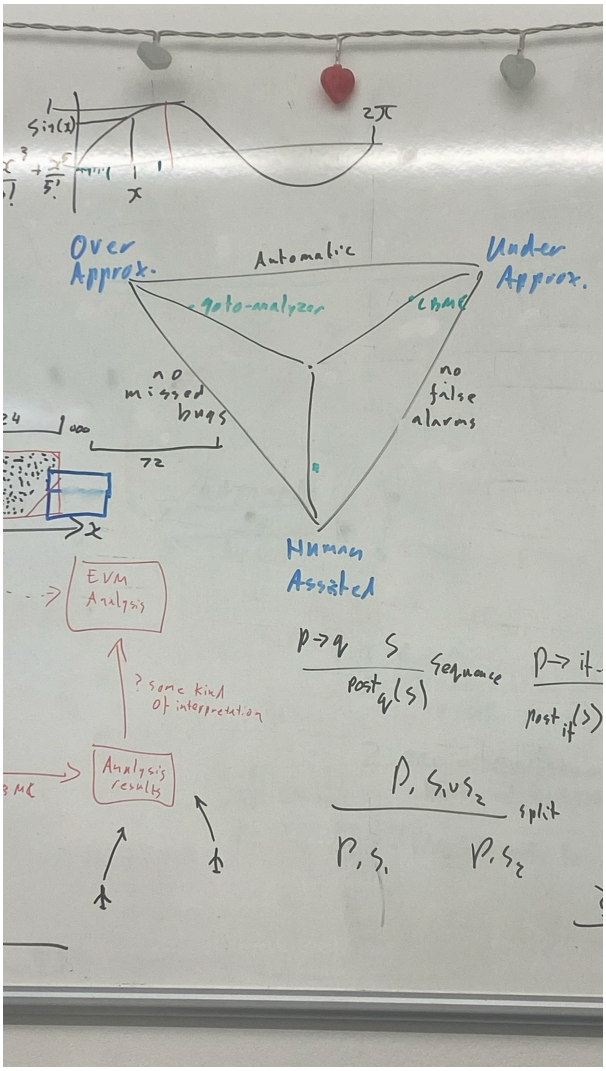
I'd start with Ethereum documentation and the Ethereum Yellow Paper

17th Feb meeting with Martin:
Decompiling .net is also more relevant to our project
Literature on Decompiling, look more at JVM Decompilation, will be closer technically (decompiling from binary i.e executables) with JVM we have types and boundaries.
State of the art for elm or for solidity , what else is out there for that top link. Look at verification of EVM and Solidity ("competitors" of my approach) that directly do verification.
Existing systems (disassembles vs full decompilation that return source code)
Approved PDD

28 Feb meeting : Design section should include:
How many intermediate formats are we using?
WHAT PARSES ARE WE DOING IN THE DATA AND HOW DO WE STRUCTURE THEM?
parsers, one that outputs it in the same format as the input, demonstrate that it works both ways
3rd parse: start trying to map to c , directly from disassembly
Tree structure it? CFG Recovery? Am I going to take my arrays and completely convert them? Control flow graph? When I print stuff out will it use the control flow graphs.
More of a Research project rather than class design
Getting the end to end, getting the intermediate formats right and the parsing correct.
Started with x , ended with Y
If there's an unrecognised op code
2nd parse: Human readable (like the table)
Parse from disassembled to c , might not be the best or most useful output, but we will find this out earlier rather than later. Inform us on what Decompilation activities we want to do.
1st parse: Parsers written using visitor patient ,(compiler internals) opcode. Java has a visit method that allows you to pass something
Iterate over the array, make a switch when you come to an instruction. Explicit or using visitor function.

10th march
For example, when processing byte code like "ADD", how do you deal with the values you are adding?

At the moment: Function where you feed in the string, from that you produce an array of objects of type op code - modify, so that instead it prints the semantics of each op code (research)
Check buildexample.txt as a way to improve the code!
Parse in , generate it out in human readable



24th March
Add Automated testing
The Halting problem for normal programs - you cannot write a program that can take in an arbitrary Turing machine and tell you whether it will ALWAYS terminate
Relitavely easy to prove , come up with an algorithm that works for some Turing machines (its easy to prove some Turing machines terminate) - same applies to programs: if given a program with a loop, how do you know it will terminate?
Wind up with a pyramid (pyramid of verification, you can have 2 sides, but to have all 3 is difficult , e.g a verification tool that is automatic and has no missed bugs, might give false alarms (for example warning messages on visual studios- shows EVERY possible problem that could occour):

(see email from this date with example code and using prover)
See email from Martin for documents , and commands that were run on the prover tool, with the c file.

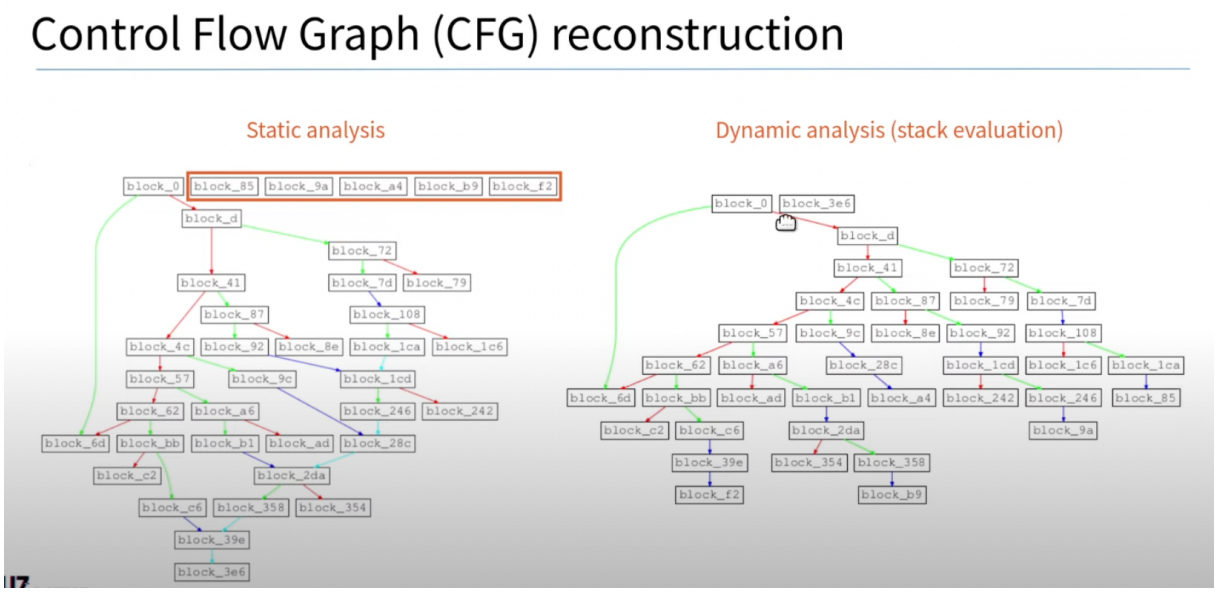
14th April:
Links for blockchain verification https://link.springer.com/book/10.1007/978-3-030-99524-9

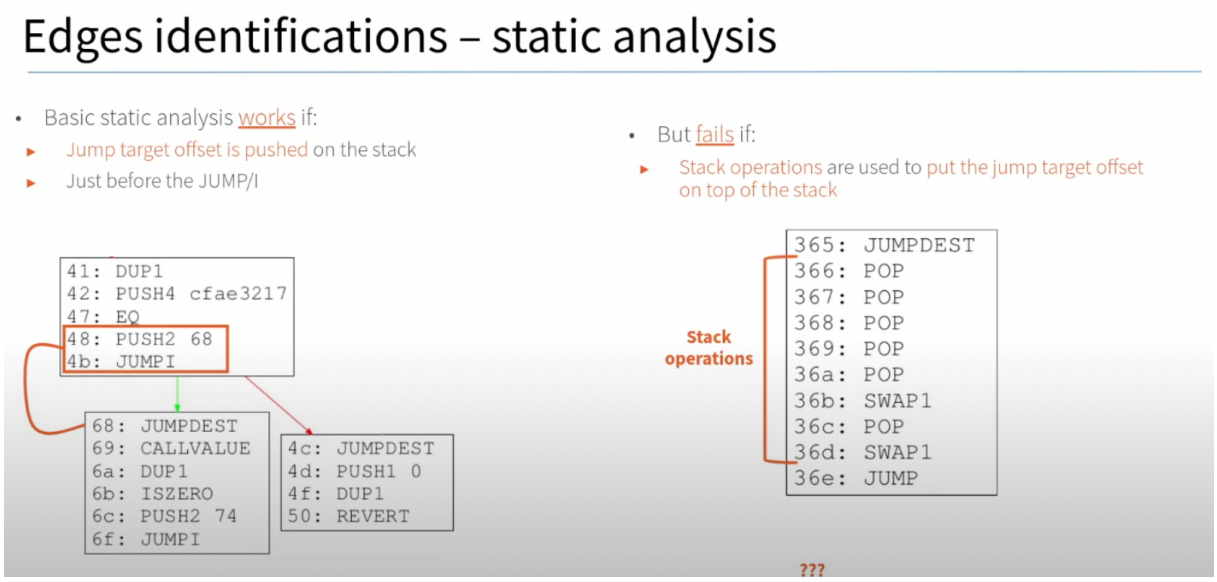
Ethereum/EVM Smart Contract Reverse Engineering & Disassembly - Blockchain Security #3
https://www.youtube.com/watch?v=I6VDBvX9Pkw&t=41s
Disassemble from EVM byte code to op code

Decompose into basic blocks of opcodes- Separate op coded based on:

Opcodes	Simplify description	Position within a Basicblock
JUMP	Unconditional jump	Last instruction
JUMPI	Conditional jump	Last instruction
RETURN , STOP INVALID SELFDESTRUCT , REVERT	Halt execution	Last instruction
JUMPDEST	Marks a position within the code that is a valid target destination for jumps	First instruction

CFG (CONTROL FLOW GRAPH) reconstruction, if you do not know the jump location, then you must use dynamic analysis rather than static, the example bellow shows that if we have the location then we can track the flow of the program:





Function identification and function names: based on the cfg constructed we can identify functions

https://ethereum.stackexchange.com/questions/188/how-can-you-decompile-a-smart-contract

Compilation back to the original source code is impossible because all variable names, type names and even function names are removed. It might be technically possible to arrive at some source code that is like the original source code but that is very complicated, especially when the optimiser was used during compilation. I don't know of any tools that do more than converting bytecode to opcodes.
Since contracts can access their own code and thus (ab)use the code for storing data, it is not always clear whether some part of the code is used as code or only as mere data and whether it makes sense to try and decompile it. It is computationally undecidable whether some piece of the code is reachable or not.
Note that there is no dedicated area to store creation-time fixed data (like lookup tables, etc). Apart from the code of the contract, it would also be possible to store the data in storage, but that would be way more expensive, so putting such data in the code is a common thing.

Note: why do I write that you match on opcodes and not instructions (i.e. opcode and operand pair)? This is because if you are doing pattern matching you want to abstract a little bit; using the opcode for the instruction does that. In those cases where operand information should be included, what is done in the Python decompiler is that the opcode changes to reflect this additional piece of abstraction. There is nothing that dictates that you must match on existing EVM opcodes. You can make up new opcodes, insert opcodes that might indicate control structure boundary or change some opcode names to assist in pattern matching.

——28/04—
See example .c file, simulate the EVM stack first, then try to reconstruct functions, variables, etc…
Visitor , for every instruction

--Last meeting before submission 09/05 ---
Var needs to be a 256-bit integer : int64_t for executable , int128_t for compliable/analysable (have a switch between an executable and analysable , give an option to make runnable code vs Cprover code)

For unsignable, call them uint64_t, #include <stdint.h>
Interactive Commands with stack
1. work out how the values of the program counter correspond to the instructions
2. Create a map between program counter values / byte instructions and labels in the C.

[2.1. Catch and error on cases when a jump would take you into the middle of a multi-byte instruction.]

[2.2. Use the map to implement the PC instruction.]

3. Be able to identify whether the top element of the stack is a constant or not when a jump is called. It is fine to just handle the simple cases of this.

4. If you are jumping to a constant location, then you can use the compile time map to work out which label to goto

The map in 2 is working out which locations it is valid to jump to. This is what is described in section 9.4.3 in the yellow paper.

5. extend 4 so that conditional jumps are _if (stack[1]) { goto ___ ; }_

(Sorry conditional jumps to known fixed instructions!)

Handling computed / non-constant jump targets is Hard. IF you do it, you should make a big deal about it in your write up.
Options...

6.A. Each time you come to a jump that you can't show to be to a constant location, generate code that uses a switch statement and go-tos for each possible location to handle it.

Using the map you built in 2...

```
switch (stack[top]) {
        case 0 : goto label_000;
        break; // no case 1 as 0 is a 2-byte instruction
        case 2 : goto label_002;
        break; .
}
```

SHA3:
Put in an existing implementation of SHA 3 (have a switch for either a computable vs analysable interpretation)
Uninterpreted function is best for an analysable solution (for Cprover only), output is non-deterministic, documentation for uninterpreted functions: http://www.cprover.org/cprover-manual/modeling/nondeterminism/)
example:
https://github.com/diffblue/cbmc/blob/develop/regression/cbmc/uninterpreted_function/uf2.c

A NOTIFICATION (Publish/ subscribe system that lets you know when things happen, the log writes to some notion of output)
Since contracts are passive, this is not as useful to implement so we simply use an assert so that we can at least show that the client is being “shown” something

- [x] LOGx: Append a log record with x topics, where x is any integer from 0 to 4 inclusive, can be a Simple assert statement: assert(0);
- [x] CREATE: Create a new account with associated code
Map of account number and a balance (pairs like the memory)

Scope meeting:
out of scope : “Message-call” and calls, and block commands- only to smart contract accounts, we have accounts that are limited to balance, this would require modelling of states, which requires a separate hash-map.
Block is out of scope, as you need the storage of every modification