nullchecks

There are mainly 4 instructions that may lead to segmentation faults while dereferencing null pointers.

1. Store
2. Load
3. GetElementPtr
4. Call (Indirect)

> **Naïve solution:** Adding nullchecks before every instructions of these four types. But it will lead to multiple compare and conditional branch instructions.

Better solution for this is to use forward data flow analysis to track every pointer and add nullchecks only when we are unsure of it being null safe. *(Similar to constant propagation data flow analysis)*

For this we have 2 maps:

- `std::unordered_map<Value *, std::unordered_set<Value *>> childPointers;`
- `std::unordered_map<Value *, bool> unsafePointers;`
- `childPointers` maps each pointer to a set of its child pointers. Or in other words every child pointer's value is dependent upon its parent pointer. For example:

  ```
  %a = load i32*, i32** %a.addr;
  ```

  In this example `%a` is the child pointer of `%a.addr`. As if the value of `%a.addr` changes the value present in `%a` is corrupted and new value may not be null safe.

- `unsafePointers` just tells whether a pointer present in the map is unsafe or not.

> The `childPointers` map is preserved through out the function, but the `unsafePointers` map resets at each basic block. As from any basic block, one can reach any basic block. Thus any basic block could have modified the pointer thus making them unsafe, so we can't rely on previous observations.

Forward data flow analysis

- There are two APIs first is `includeInst` and `processInst`.
- `includeInst` takes in the instruction and both maps. And outputs whether we should add nullchecks for this instruction.
- `processInst` adds nullchecks for the given instruction.
- The transfer functions are present in the `includeInst`.

Transfer function

> Present in the `NullChecks.cpp` from L81-L112.

1. Store instruction: When storing a value to a pointer it will make the pointer itself and all the child pointers unsafe. As we might have stored null into `%a.addr` thus making it unsafe when using it to load `%a`.
2. Load instruction: We'll first add the value operand of the `basePointer` i.e. the loaded value to the `childPointers`. Then as the loaded pointer is always safe, we'll write that fact into `unsafePointers`.
3. GetElementPtr instruction: Any pointer returned by this instruction will be checked for nullchecks when it is used. And the fact about it being null space will only be stored then.
4. Call (indirect) instruction: We'll just make the function pointer safe and `basePointer` in this case is just the function pointer itself. Reasoning behind making the function pointer safe is explained later.
5. Alloca & Icmp instruction: For these instruction, we don't need to do anything as we are not dereferencing any pointers.

6. Bitcast instruction: Same as alloca instruction we don't need to do anything as we are just type casting the pointer not dereferencing it.
7. Branch instruction: This clears the `unsafePointers` map. *Reasoning explained above...*
8. Ret instruction: This again does not do anything.

> Adding nullchecks for this instruction will depend upon `unsafePointers[basePointer]`; if the `basePointer` i.e. `%a.addr` present in the `unsafePointers` then the value itself otherwise the `basePointer` is unsafe by default.

> Also before returning we will always make the `basePointer` mark as safe. Reasoning is that if the current instructions executes after getting through the nullchecks, it means that the pointer is not null. And is safe to use for the coming instruction using that pointer.

> The above two operations does not apply for instructions where we does not do anything (alloca, icmp, bitcast, br, ret). We'll never add nullchecks for these instructions.

Meet operator

This does not do anything as `childPointers` is preserved through out the function. And `unsafePointers` is empty at the start of each basic block, or in other words it also does not depend upon the incoming `unsafePointers` map from predecessor basic blocks.

Initialization

At the start of function `childPointers` is 0 initialized and `unsafePointers` is also 0 initialized at the start of each basic block.

Assumptions

1. Return type of any function is one of these `void`, `int`, `int *`. As the pass is adding default return values as the transmission instructions for the `NullBB` *(exhaustive declaration of return instructions).*
2. Pointers passed in as function parameters are not taken in account of. Alternatively it is assumed that the function will not modify the pointers that are passed to it.
3. This pass is exhaustively tested on the given 8 test cases and may fail on anything outside the scope of those test cases.

Miscellaneous

- Outputs of all tests are compiled into `nullchecks.out`.
- All the instructions that requires nullchecks are the first instruction of every `NotNullBB`. LLVM IR files are present inside `nullchecks_opt_ll`.