| galois |

Memory Safety Proofs in SAW

What is Memory Safety?

- Memory safety is concerned with software being free from memory access errors
- Examples of memory safety violations:
  - Buffer overflows
  - Dangling pointer dereferences
  - Null pointer dereferences
  - Double frees
- SAW can detect all of the previous examples and more

Why Prove Memory Safety?

- Memory safety bugs are common
  - Easy to accidentally add
  - Hard to find
- Memory safety bugs are at the heart of many security vulnerabilities
  - Pick almost any Google Project Zero writeup. Almost all contain an exploit of a memory safety vulnerability at some step
- Why not just use Valgrind or AddressSanitizer?
  - Those tools only detect violations encountered in runtime testing
  - SAW detects memory safety bugs in *all* possible execution traces

Example: Add

```c
uint32_t add(uint32_t x, uint32_t y) { return x + y; }
```

```
let add_spec = do {
    // Create fresh variables for `x` and `y`
    x <- llvm_fresh_var "x" (llvm_int 32);
    y <- llvm_fresh_var "y" (llvm_int 32);

    // Invoke the function with the fresh variables
    llvm_execute_func [llvm_term x, llvm_term y];

    // The function returns another 32 bit value at a different memory
    location from `x` and `y`
    ret <- llvm_fresh_var "ret" (llvm_int 32);
    llvm_return ( llvm_term ret );
};
```

Declaring a Function

```
let fn_name arg1 arg2 = do { … };
```

- Declare functions using `let`
- In many cases, you do not need any arguments.
    - Arguments are useful for specifying functions that can take inputs of differing sizes (we will get there later)
- Important: Function declarations must end in a ‘;’
    - You will forget to do this. I still make this mistake. You will get a nasty error.
- Let bindings also used for variables

Comments

- C style comments
- // for line comments
- /* … */ for block comments

Fresh Variable Declarations

```
x <- llvm_fresh_var “x” (llvm_int 32);
```

- Use llvm_fresh_var to declare a symbolic variable
  - Symbolic variables can hold any value
- First argument is a string to show during debugging
  - Best practice: Match name with variable in C source
- Second argument is the type the variable has
  - 32-bit integer in this case
  - We will cover other types later

`let` vs `<-`

- SAWScript distinguishes between defining a name and saving the result of a command.
- Use `let` to define a name, which may refer to a command or a value
- Use `<-` to run a command and save the result under the given name
- Defining a command with `let` is analogous to defining a C function
- Invoking commands with `<-` is analogous to calling it

Specifying Function Invocation

llvm_execute_func [args];

- `llvm_execute_func` instructs SAW how to invoke the function under verification
- Takes a list of function arguments
- Going back to our add example:

```
// Invoke the function with the fresh variables
llvm_execute_func [llvm_term x, llvm_term y];
```

- Use `llvm_term` to convert the SAW type `llvm_fresh_var` returns into a C value

`llvm_execute_func` Placement

{preconditions}
llvm_execute_func [args];
{postconditions}

- SAW treats everything above `llvm_execute_func` in a spec as the function's preconditions
- SAW treats everything below `llvm_execute_func` in a spec as the function's postconditions
- SAW verifies: assuming everything above `llvm_execute_func` is true, then everything below `llvm_execute_func` must be true

Specifying Return Values

```
llvm_return (llvm_term ret);
```

- Use `llvm_return` to specify the return value of a function
- Takes one argument containing the returned value.

Putting it All Together

```
let add_spec = do {
    // Create fresh variables for `x` and `y`
    x <- llvm_fresh_var "x" (llvm_int 32);
    y <- llvm_fresh_var "y" (llvm_int 32);

    // Invoke the function with the fresh variables
    llvm_execute_func [llvm_term x, llvm_term y];

    // The function returns another 32 bit value at a different memory
    location from `x` and `y`
    ret <- llvm_fresh_var "ret" (llvm_int 32);
    llvm_return ( llvm_term ret );
};
```

Exercise: Popcount Spec

- Look at memory-safety/popcount/popcount.c
  - Contains 3 definitions of a function that count all of the set bits in a 32-bit integer
  - Note that all functions have the same signature (take a `uint32_t`, return an `int`).
  - Assume `int` is 32 bits.
- Complete Part 1 in memory-safety/popcount/exercise.saw
- Feel free to use the files in complete_examples/add/ as a guide.

Verifying Add Example

```
// Load LLVM bitcode to verify
m <- llvm_load_module "add.bc";

// Verify the `add` function satisfies its specification
llvm_verify m "add" [] true add_spec z3;
```

```
[23:00:38.061] Verifying add ...
[23:00:38.075] Simulating add ...
[23:00:38.076] Checking proof obligations add ...
[23:00:38.076] Proof succeeded! add
```

llvm_verify

```
llvm_verify <module> “<C function name>”
    [<overrides>] <check path conditions?>
    <SAW spec> <proof script>
```

- `<module>` - Loaded LLVM bytecode
- `<C function name>` - C function to verify against
- `[<overrides>]` - List of overrides
  - Enables compositional verification as we’ll see later
- `<prune paths?>` - Check that paths are reachable before exploring them
  - In practice always set this to `true`
- `<SAW spec>` - SAW specification to verify code against
- `<proof script>` - Strategy SAW should use to check verification conditions
  - For now, use `Z3` or `Yices`. We will talk about other options tomorrow.

Exercise: Popcount Proof

- Complete Part 2 in memory-safety/popcount/exercise.saw
- Feel free to use the files in complete_examples/add/ as a guide.

Pointers

Create a variable with a name and a size

```
x <- llvm_fresh_var "x" (llvm_int 32);
```

C analog: `int x;`

Allocate a memory block, returning a pointer

```
xp <- llvm_alloc (llvm_int 32);
```

C analog: `int* xp;`

Assert that a pointer points to a value

```
llvm_points_to xp (crucible_term x);
```

C analog: `xp = x;`

Run the program with arguments

```
llvm_execute_func [xp];
```

pointer_to_fresh Helper Function

```
/**
 * Creates a fresh symbolic variable with name of specified type,
 * initializes the pointer to the variable's location, and returns
 * the tuple (variable, pointer)
 */
let pointer_to_fresh (type : LLVMType) (name : String) = do {
    x <- llvm_fresh_var name type;
    p <- llvm_alloc type;
    llvm_points_to p (llvm_term x);
    return (x, p);
};
```

Exercise: Swap Spec

- Look at memory-safety/swap/swap.c
    - Look at `swap` and `xor_swap`
    - Both functions swap values at different pointers using different techniques.
- Complete Parts 1-4 in memory-safety/swap/exercise.saw

Arrays

llvm_array <size> <type>;

- Use llvm_array with the existing constructs we've already talked about to support functions that take arrays as arguments.
- Example: To create a pointer to an array of 16 32-bit ints:

```
(a, a_ptr) <-
   pointer_to_fresh (llvm_array 16 (llvm_int 32)) “a”;
```

- Problem: What if we don't want to fix an array size in our spec?

Parameterized Specs

- To write specs for functions that support multiple input sizes, add a size parameter to your SAW spec

Example: Given a C function:

```
some_fn(uint32_t* a, uint32_t size);
```

Partial spec for `some_fn`:

```
let some_fn_spec size = do {
  (a, a_ptr) <-
    pointer_to_fresh (llvm_array size (llvm_int 32)) “a”;
  llvm_execute_func [a_ptr,
                     llvm_term {{ `size : [32] }}];
   …
};
```

Using Parameterized Specs

- You will need to provide a concrete value when you use a parameterized spec.
- For example, to prove `some_fn` is correct for a few different input sizes:

```
llvm_verify … "some_fn" … (some_fn_spec 4) …;
llvm_verify … "some_fn" … (some_fn_spec 12) …;
llvm_verify … "some_fn" … (some_fn_spec 256) …;
```

Exercise: Selection Sort Memory Safety

- Quick selection sort refresher:
  - Sorts an array in $O(n^2)$ time
  - Scans the entire array for smallest element and swaps with the front of the array.
  - Then, scans from index 1 for the next smallest element and swaps with the value at index 1
  - And so on until it has looped through entire array
- Look at `selection_sort` in `swap.c`
- Complete exercises 5 and 6 in memory-safety/swap/exercise.saw

Proof Composition

- Can save the result of one proof (called an override) and use it in another!
  - Greatly improves performance as SAW does not need to revisit functions it has already visited.
- Assume the function `foo` calls `bar`, and we have a proof for bar:

```
bar_ov <- llvm_verify … "bar" … bar_spec …;
llvm_verify … "foo" [bar_ov] … foo_spec …;
```

- When SAW encounters a call to `bar` during the verification of foo, it:
  1. Checks that the preconditions for `bar` hold at the callsite
  2. Replaces the function call with `bar_spec`'s postcondition

Proof Composition Notes

- Always use proof composition when you already proved a function called by another function you are verifying
- When you run into performance problems, break code apart into functions, prove them separately, and make use of proof composition in the calling function.
- Loop bodies are great candidates for breaking into helper functions and proving separately.
- You can make use of multiple overrides for the same function in a proof.
    - SAW will pick the right one by looking at preconditions
    - For example, our swap specs with same or different pointer arguments are mutually exclusive. SAW always knows which one to dispatch.

Common Commands in SAW

Create a variable with a name and a size

```
x <- llvm_fresh_var "x" (llvm_int 32);
```

Allocate a memory block, returning a pointer

```
xp <- llvm_alloc (llvm_int 32);
```

Run the program with arguments

```
llvm_execute_func [xp];
```

Assert that a pointer points to a value

```
llvm_points_to xp (llvm_term x);
```

Perform the verification

```
llvm_verify <llvm module> "<function name>"
    [<overrides>] true <spec name> abc;
```

Exercise: Selection Sort Proof Composition

- Complete exercises 7 and 8 in memory-safety/swap/exercise.saw

Arrays as Integers

- In some cases, it may be convenient to represent an array as a single integer.
    - For example, when representing a 128-bit int in C as an array of 2 64-bit ints.
- To support this, llvm_int supports arguments other than the “standard” int sizes, including sizes over 64 bits.
- Ex: The C type uint16_t[3] can be represented in SAW as:
    - llvm_array 3 (llvm_int 16), or
    - llvm_int 48

Exercise: `increment_u128`

- Look at `increment_u128` in memory_safety/u128/u128.c
- Complete part 1 of the exercise in memory_safety/u128/exercise.saw

Exercise: A Strange Error…

- Take a look at part 2 of the u128 exercise
- We want to verify `eq_128`, which uses bcmp to check if `x` and `y` are equal
- We have a spec for `eq_128` that looks correct
- Uncomment the `llvm_verify` command. Why do you think this fails?

Answer: Dynamic Linking

- Libc is dynamically linked. `u128.bc` does not contain a definition of `bcmp`
- SAW needs overrides for functions in dynamically linked libraries
- SAW has some overrides built in for commonly used functions (like `malloc`), but not for *all* libc functions, nor for any other libraries your code may link against.

Assuming Overrides

foo_ov <- llvm_unsafe_assume_spec m “foo” foo_spec;

- Use `llvm_unsafe_assume_spec` to assume an override
- `m` - the loaded LLVM bitcode containing `foo`
- `“foo”` - the C function to assume an override for.
- `foo_spec` - the specification you wish to assume
  - Write these specs just like the other specs we’ve been writing so far
- Use `foo_ov` in the override list for any function you’re verifying that calls `foo`
- With great power comes great responsibility: *Really* check over any overrides you assume. SAW does not check these.

Exercise: Override bcmp

- Assume an override for bcmp and fix the proof in part 2 of the u128 exercise.

Preserving Pointer Values

- Remember: when using an override, function call is replaced with the override's postcondition
- Therefore, we need to specify the values of all pointers after `llvm_execute_func`
- In previous example, could do this with:

```
(x, x_ptr) <- pointer_to_fresh (llvm_int 128);
…
llvm_execute_func [x_ptr, y_ptr];
llvm_points_to x_ptr (llvm_term x);
…
```

An Easier Way: Readonly Values

```
/**
 * Creates a fresh, read-only, symbolic variable with name and type,
 * returns tuple (variable, pointer)
 */
let pointer_to_fresh_readonly (type : LLVMType) (name : String) = do {
    x <- llvm_fresh_var name type;
    p <- llvm_alloc_readonly type;
    llvm_points_to p (llvm_term x);
    return (x, p);
};
```

Exercise: eq_u128 Readonly Arguments

- Complete part 3 of the u128 exercise

Structs

`llvm_struct “struct.<name>”`

- Much like ints with `llvm_int`, and arrays with `llvm_array`, SAW supports struct types with `llvm_struct`.
  - Can use `llvm_struct` anywhere SAW expects a C type.

Ex: Given a C struct:

```
struct options { … };
```

The corresponding SAW type is:

```
llvm_struct “struct.options”
```

Exercise: Point Structs

- Look at memory-safety/point/point.c
  - Library for creating, copying, adding, and checking the equality of 2d points
- Complete part 1 of exercise in memory-safety/point/exercise.saw

Global Variables

- In some cases, SAW can handle global variables without help
  - Const global variables defined in LLVM bitcode file you're verifying
- In other cases, you will need to define global variables so SAW knows what value they should have during verification.
  - You can also define global variables to be symbolic
- Declare a global: `llvm_alloc_global "<name>"`
  - `<name>` must exactly match the name of the global variable
  - Allocates the global's memory in SAW
- Get a pointer to a global: `llvm_global "<name>"`
  - `<name>` must exactly match the name of the global variable

Global Variables: Example

```
bool FLAG;
void fn(...) {
    if (FLAG) {...}
}

let fn_spec = do {
    llvm_alloc_global “FLAG”;
    llvm_points_to (llvm_global “FLAG”)
                   (llvm_term …);
    …
};
```

Exercise: Global Variables

- Complete part 2 of the point exercise

Best Practice: Prove Memory Safety First

- Memory safety is often easier to prove than functional correctness
  - Frees you up to think about proof structure
- Functional correctness proofs necessarily include memory safety
  - SAW still checks memory safety conditions in functional correctness proofs
  - Fewer places proof errors can come from when writing memory safety proofs

What have we learned?

- How to write memory safety proofs in SAW!
  - A ton of SAW commands for specifying memory layout
  - Proof structure / composition
- Memory safety proofs are relatively low effort but cover a wide range of common bugs
  - In some cases you may determine that memory safety is sufficient for a program

Questions?