Formal Methods in Computer-Aided Design 2024

Ownership in low-level intermediate representation

Siddharth Priya
University of Waterloo
Waterloo, Canada
siddharth.priya@uwaterloo.ca

Arie Gurfinkel
University of Waterloo
Waterloo, Canada
arie.gurfinkel@uwaterloo.ca

***Abstract*—The concept of ownership in high level languages can aid both the programmer and the compiler to reason about the validity of memory operations. Previously, ownership semantics has been used successfully in high level automatic program verification to model a reference to data by a first order logic (FOL) representation of data instead of maintaining an address map. However, ownership semantics is not used in low-level program verification. We have identified two challenges. First, ownership information is lost when a program is compiled to a low-level intermediate representation (e.g., in LLVM IR). Second, pointers in low-level programs point to bytes using an address map (e.g., in unsafe Rust) and thus the verification condition (VC) cannot always replace a pointer by its FOL abstraction. To remedy the situation, we develop ownership semantics for an LLVM-like low-level intermediate representation. Using these semantics, the VC can opportunistically model some memory accesses by a direct access of a pointer *cache* that stores byte representation of data. This scheme reduces instances where an address map must be maintained, especially for mostly safe programs that follow ownership semantics. For unsafe functionality, memory accesses are modelled by operations on an address map and we provide mechanisms to keep the address map and pointer cache in-sync. We implement these semantics in SeaBMC, a bit-precise bounded model checker for LLVM. For evaluation, the source programs are assumed to be written in C. Since C does not have ownership built-in, suitable macros are added that introduce and preserve ownership during translation to LLVM-like IR for verification. This approach is evaluated on mature open source C code. For both handcrafted benchmarks and practical programs, we observe a speedup of 1.3x–5x during SMT solving.**

I. Introduction

Ownership is a scheme to control aliasing of references in high level languages. It has been studied in a long line of academic research [1], [2], [3], [4]. More recently, the concept has gained attention due to Rust, a popular systems language that offers low level control like C/C++ and uses ownership semantics to record aliases and mutation of data. In Rust, (1) a value has exactly one owner, (2) a reference to a value (called a *borrow*) cannot outlive the owner, and (3) a value can have one mutable reference *xor* many immutable references. A program that follows this programming discipline allows the Rust compiler to reason about memory safety statically. However, for reasons of expressivity and performance, programs may need to break this discipline for certain operations. For this, Rust provides *unsafe* code blocks where the static checks are temporarily turned off.

While, ownership can aid in generating correct and efficient code, it is also useful in program verification. Usually, the presence of aliasing necessitates an *address map* to soundly model object accesses through different aliases. With ownership, this map can be eliminated when it is known that only a single reference exists. This has been useful for program verification. For example, the Move Prover [5] replaces references by objects in the generated verification conditions (VC). Similarly, RustHorn [3], [6] is able to generate pure First Order Logic VC for safe Rust programs without introducing a memory model.

The advances in verification using ownership semantics have not made their way to verification of low-level programs. One of the problems is that low-level languages do not support ownership out of the box. As an example, LLVM bitcode is a register based intermediate representation (IR) used by C, C++, and, Rust compilers. It only has an attribute for marking pointers as `noalias` [7] and no ownership operations. The `noalias` attribute is useful for optimization. However, the semantics of `noalias` in unclear and has caused confusion [8]. Another challenge is that ownership in high-level language does not translate directly to low-level settings. For example, in verification of safe Rust programs, it is correct to model a reference by the FOL representation of the value it refers to. However, this model does not work for LLVM-like IR (and unsafe Rust) because such languages (dialect) have pointers that treat values as a collection of bytes and rely on pointer arithmetic to access individual bytes. In verification, the standard solution models memory using an address map from addresses to byte or word values. However, such address maps are expensive to execute symbolically.

This work improves the state-of-the-art by the following contributions. First, we develop an ownership semantics for an LLVM-like low level language that operates on single words in memory. This language replaces unrestricted aliasing with mutable borrow, read-only borrow, and copy operations that track outstanding aliases for a memory allocation. Second, we define a caching mechanism for capturing data at a pointer itself. This cache can by written and read by operations on the pointer. A pointer cache allows us to replace memory accesses in the generated VC by the cache whenever correct to do so. This can simplify the VC and improve the solving time. For mostly safe programs, many memory accesses may be replaced by pointer cache accesses. These semantics are discussed in Section II.

Third, we discuss our design for VCGen and especially modelling the borrow operation in Section III. Borrowing temporarily transfers memory access rights from the lender pointer (a.k.a. *lender*) to the borrowing pointer (a.k.a. *borrower*). In

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_35

the given semantics, this means that the pointer cache is also copied from lender to borrower. However, the borrower is assumed to return the borrow by the first instance of a memory access by a lender. This means that the updated cache at the borrower *must* be copied back to the lender before this. This transfer ordinarily requires a memory model that allows shared accesses between the borrower and lender aliases. A more efficient way uses prophecy variables [9] and was first proposed in [3]. We adapt the prophecy solution to our setting.

Fourth, to make our ownership semantics practical, we add support for multi-word memory operations and evaluate the semantics by incorporating it in SEABMC [10]. SEABMC is a bit-precise bounded model checking engine for SEAHORN that uses an SMT solver as its backend. To ease writing programs using these ownership semantics, the user writes C programs laced with calls to ownership macros. During compilation, the macros expand to LLVM intrinsics that are then interpreted using the given semantics to generate VC. The benchmark programs are a mix of handcrafted examples and practical programs. The handcrafted examples are used to fine tune performance and show what is possible. The practical programs are from the `mbedTLS` project - an open-source SSL/TLS library and has routines for encryption and secure communications. We get a speedup of 1.3x–5x during SMT solving. We see that the verification simplicity (speedup) correlates positively with the number of memory accesses that can be replaced by pointer cache accesses in a program.

II. OSEA-IR LANGUAGE

In this section, we present the syntax and semantics of the OSEA-IR language. To simplify the presentation, we propose machines that work with a single datatype $bv(N)$, a bit-vector of N bits, as the *word* size. We impose two restrictions. First, all memory operations - *allocation*, *load*, and, *store* work on a single word. Second, the machine can only store integers in memory and does not support *store* and *load* of pointers. We lift the first restriction in Section IV, and the second in an extended version [11].

Syntax. We introduce ownership semantics on the base language SEA-IR [10]. SEA-IR is an intermediate representation (IR) itself based on LLVM IR. LLVM assumes a register based machine and dependency between memory operations are implied. SEA-IR explicates this dependency information between memory operations by introducing memory registers. We assume that the type of each register is known. Figure 1 shows the ownership extended syntax of SEA-IR called OSEA-IR. We use `R` to represent a scalar register, `P` for a pointer register and `M` for a memory register. A legal OSEA-IR program is assumed to be in a Static Single Assignment (SSA) form. OSEA-IR primarily replaces unrestricted alias creation by new operations that introduce and remove aliases in a restricted manner. The `mk_own` instruction initializes memory at the given location (simlar to a `Box::new(n)` in Rust). The `mut_mkbor`, `mut_mksuc` instructions occur in pairs. The first creates a mutable borrow pointer from a lender pointer. The second creates a succeeding pointer from the lender pointer that becomes active after the mutable borrow ends. The `mut_mkbor_off` is similar to a `mut_mkbor` and creates a pointer at an offset within an allocation. It must be followed by a `mut_mksuc` instruction. The `ro_*` instructions create read-only borrows of the lending pointer. The `cpy_*` instructions create unrestricted copies of the lending pointer. The `mut_mkbor_mem2reg` instruction borrows (loads) a pointer stored in memory to a register. The `mov_reg2mem` instruction moves (stores) a pointer in a register to memory. There is no move instruction between registers since the operation is equivalent to α-renaming.

Semantics of $\mathcal{M}_0$. The semantics are given in terms of a machine $\mathcal{M}_0$ and is based on the stacked borrows model for Rust [12]. In our formulation, each pointer type ($ptype$) is one of owned (o), mutably borrowed (mb), immutably borrowed (rb), or, copied (c). Access to memory is controlled by maintaining a per-location borrow stack that captures both valid accessors and access order. The configuration of $\mathcal{M}_0$ is given by the program counter state (P), register map (R : id $\rightarrow$ $value$), address map (M : $addr \rightarrow value$) and a borrow store state (S_B : $addr \rightarrow stack((tag, ptype))$). A $value$ is either a bit-vector $bv(N)$ or a pointer type. An address ($addr$) is represented as a bit-vector. A pointer is a tuple of $(addr, tag)$ and is considered *fat* on account of additional metadata carried along with the address[1]. A $tag : bv(N)$ is a unique id given to a pointer when it is defined. Operations that introduce and remove aliases, then push and pop alias tags on the borrow stack, respectively. Each borrow stack entry also stores $ptype$ along with an identifier for finer access control. An important restriction is that memory access is allowed for an alias if its tag is top-of-(borrow)stack for that address.

The semantics for relevant pointer introduction, aliasing, and, removal are given through operations on the borrow stack (B) in Table I. A borrow stack state is represented as a list $B = e :: B_1$, where e is the top of stack and B_1 represents the rest of the stack. We do not explicitly show effect of operations on (P, R, M) nor do we give the semantics for all instructions of $\mathcal{M}_0$ due to space constraints. The interested reader is referred to the stacked borrows [12] and SEA-IR [10] papers for further background. The `mk_own` operation allocates and stores n at the given location. This operation must provide a location that is un-allocated. After the operation, the new pointer tag is pushed onto the stack with $ptype =$ o. The mutable borrow operations use `mut_mkbor`, `mut_mksuc` instructions that always occur in a pair on a lender pointer p_0 to create a borrowed pointer q_0 and a succeeding pointer p_1. For a successful operation, the borrow stack is popped until p_0 is on top and its $ptype$ is either an owning or a mutably borrowed pointer. This operation removes $p_0.tag$ as an accessor and instead pushes $p_1.tag$, the succeeding pointer and $q_0.tag$, the borrowed pointer, to the borrow stack in that order. The associated type of a pointer is also added to each stack entry. Note that the type of q_1 is always mb. However

---

[1] We use the shorthand $.addr$ to refer to the first tuple element, similarly for other elements.

```
   ⟨S⟩     ::=   ... | ⟨OS⟩
⟨RDEF⟩     ::=   ... |
                 ⟨P⟩, ⟨M⟩ = mk_own ⟨R⟩, ⟨M⟩
                 ⟨P⟩ = mut_mkbor ⟨P⟩  |  ⟨P⟩ = mut_mkbor_off ⟨P⟩,  ⟨R⟩  |  ⟨P⟩ = mut_mksuc ⟨P⟩  |
                 ⟨P⟩ = ro_mkbor ⟨P⟩  |  ⟨P⟩ = ro_mkbor_off ⟨P⟩,  ⟨R⟩  |  ⟨P⟩ = ro_mksuc ⟨P⟩  |
                 ⟨P⟩ = cpy_mkcpy1 ⟨P⟩  |  ⟨P⟩ = cpy_mkcpy1_off ⟨P⟩,  ⟨R⟩  |  ⟨P⟩ = cpy_mkcpy2 ⟨P⟩  |

⟨MDEF⟩     ::=   ... | ⟨P⟩,  ⟨M⟩ = mut_mkbor_mem2reg ⟨P⟩,  ⟨M⟩  |  ⟨M⟩ = mov_reg2mem ⟨P⟩,  ⟨P⟩,  ⟨M⟩
  ⟨OS⟩     ::=   die ⟨P⟩
```

Fig. 1: Ownership instr. in OSEA-IR grammar, where `R`, `P`, and `M` are scalar registers, pointer registers, and memory registers respectively.

Operation	**Pre-condition**	**Post-condition**
`p,m1 = mkown n, m0`	$S_B[p.addr] = \varnothing$	$S_B[p.addr] = (tag_p, \mathsf{o}) :: []$
`q0 = mut_mkbor p0` `p1 = mut_mksuc p0`	$S_B[p_0.addr] = B_0 :: (tag_{p_0}, t) :: B_1,$ $t \in \{\mathsf{o}, \mathsf{mb}\}, p_0.tag = tag_{p_0}$	$S_B[p_0.addr] = (tag_{q_0}, \mathsf{mb}) :: (tag_{p_1}, t) :: B_1$
`c1 = cpy_mkcpy1 p0` `c2 = cpy_mkcpy2 p0`	$S_B[p_0.addr] = B_0 :: (tag_{p_0}, t) :: B_1,$ $p_0.tag = tag_{p_0}$	$S_B[p_0.addr] = (tag_{c_1}, \mathsf{c}) :: (tag_{c_2}, t) :: B_1$
`die q`	$S_B[q.addr] = (tag_q, t_q) :: (tag_p, t_p) :: B_1,$ $q.tag = tag_q, t_q = \mathsf{mb}, t_p \in \{\mathsf{o}, \mathsf{mb}\}$	$S_B[q.addr] = (tag_p, t_p) :: B_1$
`m1 = store r, p, m0`	$S_B[p.addr] = B_0 :: (tag_p, t_p) :: B_1,$ $t_p \neq \mathsf{rb}, p.tag = tag_p$	$S_B[p.addr] = B_2 :: (tag_p, t_p) :: B_1,$ $(p.tag = tag_p, t_p \in \{\mathsf{o}, \mathsf{mb}\}) \implies (B_2 = \varnothing),$ $(t_p = c) \implies (B_2 = B_0)$
`r = load p, m`	$S_B[p.addr] = B_0 :: (tag_p, t_p) :: B_1,$ $p.tag = tag_p$	$S_B[p.addr] = B_2 :: (tag_p, t_p) :: B_1,$ $(t_p \in \{\mathsf{o}, \mathsf{mb}, \mathsf{rb}\}) \implies (B_2 = [(tag_q, t_q) \in B_0 \mid t_q = \mathsf{c}]),$ $(t_p = c) \implies (B_2 = B_0)$

TABLE I: Effect of selected operations on borrow stack (S_B) in machine $\mathcal{M}_0$. Effects on R and M are not shown.

the type of p_1 depends on the type of p_0. The intent is for q_0 to have access rights till it surrenders them to p_1.

The copy operation creates two copies c_1 and c_2 using `cpy_mkcpy1` and `cpy_mkcpy2` instructions. A copied pointer corresponds to a raw pointer in Rust. The lender pointer p_0 for a copy operation can be of any *ptype*. Similar to a mutable borrow operation, all entries on top of p_0 are popped from the borrow stack and p_0 itself is removed. Next $c_1 :: c_2$ are pushed onto the borrow stack in that order. The *ptype* of c_1 is always c. However, the *ptype* of c_2 depends on the lender pointer p_0. This ensures that the *ptype* of a lender pointer is not lost through successive copy operations. Finally, the `die` operation surrenders access rights for a pointer by popping off its entry from the borrow stack. It is only defined for a mutably borrowed pointer q and signals transfer of data from such a pointer to its immediate lender, which must be of $ptype = \mathsf{o}$ or $ptype = \mathsf{mb}$. The pointer q must be top of borrow stack. The `die` operation is an extension of stacked borrows and is useful for returning information from a mutable borrow to the succeeding pointer without going through shared memory. The `store` instruction writes a value to memory. If the lender pointer p is mutably borrowed or owning then all elements before p are popped. If p is copied then borrow stack remains unchanged. The `load` instruction reads values from memory into a register using a lender pointer p. If p is owning, mutably borrowed or read-only borrowed, then all pointers above p (except copied pointers) are removed from the borrow stack. If p is copied, then the borrow stack is unchanged. Finally, the observable state $ObsState_{\mathcal{M}_0}$ of machine $\mathcal{M}_0$ is given by the tuple (P, R, M, S_B).

Let us look at an example of how $\mathcal{M}_0$ operates in Fig. 2. The intent of the program is to (1) create an owned pointer, (2) make its alias (3) update data through the alias, and, (4) observe the data through the owned pointer. At line 5, a word of memory is allocated with `(addr=0x4,tag=1)` in the register map at key `p0`, the integer `42` is written to memory at `M[0x4]`, and the *tag* value 1 is pushed to the borrow stack at `SB[0x4]`. Next an alias is created using the mutable borrow operation at lines 7–8 using tags `3` and `2` for borrowed `q0` and succeeding pointer `p1` respectively. First the tag for `p1` is pushed, then the tag for `q0` is pushed. The next couple of lines load `42` using `q0`, increment it, and write it back. The program ends the mutable borrow in line 14. This removes `q0`'s tag from `SB`. Now only `p1` can access *addr* `0x4`. Finally, the program reads the new value `43` from *addr* `0x4` in line 16.

Semantics of $\mathcal{M}_1$. We now define an extension to $\mathcal{M}_0$ called $\mathcal{M}_1$. In $\mathcal{M}_1$, a fat pointer additionally has a *cache* bit-vector field called val. Each store operation also updates val with the value to be written to memory. A load from memory may be replaced by val when correct to do so. A pointer value now becomes $(addr, tag, val)$. Overall, the semantics of existing instructions aim to maintain the val cache. The semantics is laid out in Table II. The `mk_own` instruction updates its cache with the value it initialized the memory allocation with. The pair of `mut_mkbor` and `mut_mksuc` operations have two cases: (1) if the lender is top-of-(borrow)stack then the operation reads the value stored at lender pointer p_0 and updates the caches of q_0 and p_1 with that value; (2) if the lender is not top of stack then the value at lender may be stale and the correct value is read from memory. The pair of `cpy_mkcpy1` and `cpy_mkcpy2`

Operation	Pre-condition	Post-condition
`p = mkown n`	–	$R[\mathsf{p}] = (p.addr, tag_p, n), M[p.addr] = n$
`q0 = mut_mkbor p0` `p1 = mut_mksuc p0`	$R[\mathsf{p_0}] = (p_0.addr, tag_{p_0}, v_p)$	$R[\mathsf{q_0}] = (p_0.addr, tag_{q_0}, v), R[\mathsf{p_1}] = (p_0.addr, tag_{p_1}, v),$ $(B_0 = \varnothing) \implies (v = v_p),$ $(B_0 \neq \varnothing) \implies (v = M[p_0.addr])$
`c1 = cpy_mkcpy1 p0` `c2 = cpy_mkcpy2 p0`	$R[\mathsf{p_0}] = (p_0.addr, tag_{p_0}, v_p)$	$R[\mathsf{c_1}] = (p_0.addr, tag_{c_1}, v), R[\mathsf{c_2}] = (p_0.addr, tag_{c_2}, v),$ $(B_0 = \varnothing \wedge t = \{\mathsf{o}, \mathsf{mb}, \mathsf{rb}\}) \implies (v = v_p),$ $\neg(B_0 = \varnothing \wedge t = \{\mathsf{o}, \mathsf{mb}, \mathsf{rb}\}) \implies (v = M[p_0.addr])$
`die q`	$R[\mathsf{q}] = (q.addr, tag_q, n)$	$R[p] = (q.addr, tag_p, n),$ $\exists p.R[p] = (q.addr, tag_p, \_)$
`m1 = store r, p, m0`	$R[\mathsf{p}] = (p.addr, tag_p, \_)$	$M[p.addr] = v, R[\mathsf{p}] = (p.addr, tag_p, v)$
`r = load p, m`	$R[\mathsf{p}] = (p.addr, tag_p, v_p)$	$R[\mathsf{r}] = v, R[\mathsf{p}] = (p.addr, tag_p, v),$ $((B_0 = \varnothing, t_p \in \{\mathsf{o}, \mathsf{mb}\}) \implies (v = v_p))$ $((B_0 \neq \varnothing \vee t_p = \mathsf{c}) \implies (v = M[p.addr]))$

TABLE II: Effect of selected operations on S_B,R, and M in machine $\mathcal{M}_1$ in addition to pre-and-post conditions from Table I.

```
fun main() {
BB0:
  m00 = mem.init()
  ;   R = [] |   M = [] |   SB = []
  p0,m0 = mk_own 42, m00
  ;   R[p0] = (0x4,1) |  M[0x4] = 42 |   SB[0x4] = 1 :: []
  q0 = mut_mkbor p0
  p1 = mut_mksuc p0
      R[p1] = (0x4,2) |     |
  ;   R[q0] = (0x4,3) |  M  |   SB[0x4] = 3 :: 2 :: []
  r1 = load q0, m0
    R[r1] = 42 |  M |  SB ;
  m1 = store r1 + 1,q0,m0
  ;   R |  M[0x4] = 43 |  SB
  die q0
  ;   R |  M |  SB[0x4] = 2 :: []
  r = load p1, m1
  ;   R[r] = 43 |  M |  SB
  halt
}
```

Fig. 2: Example of $\mathcal{M}_0$ operation. Effect on register map (R), memory map (M), and borrow store (S_B) shown in pink.

instructions similarly update the cache of c_1 and c_2 with the correct value. The `die` instruction transfers the value cached at q to the cache of the immediately succeeding pointer, called p here. The transfer to the succeeding pointer occurs by first searching for the pointer with the correct tag in the register map R and then updating the corresponding val field. Since we do not support the storage of pointers to memory, the search through R is enough to find the right pointer. Note that the `die` operation enables transfer of a value from a mutable borrow to the succeeding pointer without using shared memory. A `store` instruction updates the cache with the value `r` to be written to memory. This value is then written to memory and to $p.val$. In $\mathcal{M}_1$, a `store` does not support storing pointers to memory. This restriction is lifted in an extended version [11]. A `load` has two cases. First, if the lender pointer p is top-of-(borrow)stack, and is mutably borrowed or owning, then the read from memory is replaced by a read of the val (cache) field. Second, if the `load` uses a lender pointer p that is not top-of-(borrow)stack, or is copied, then the read from memory proceeds as usual. In the second case, the pointer cache is also updated with the value read from memory.

The optimisation we describe for the `load` instruction is correct because $\mathcal{M}_1$ always maintains the following invariant:

Theorem 1 (Cache equivalence). *For all pointers in the register map, if the pointer is top-of-(borrow)stack and is owning or mutably borrowed then the pointer cache value is the same as the value of memory at address of the pointer. Formally, let R be a register map, M memory, and S_B a borrow store. Then,*

$$\begin{aligned}(R[p] = (addr, tag_p, n)) \wedge \\ (S_B[addr] = (tag_p, t_p) :: B) \wedge \\ (t_p \in \{\mathsf{o}, \mathsf{mb}\})) \implies M[addr] = n\end{aligned}$$

Proof. The proof proceeds by structural induction on the syntax of the program P. Assume Thm. 1 holds in some configuration (P_0, R_0, M_0, S_{B_0}). The next instruction takes the configuration to (P_1, R_1, M_1, S_{B_1}). We case-split on each possible instruction. We illustrate the process through some of the relevant instructions.

- `store` keeps the cache in-sync with memory according to given semantics;
- `mut_mkbor` keeps the mutably borrowed pointer cache in-sync with memory since the lender cache value is already in-sync (by assumption) and mutably borrowed pointer cache gets this value;
- `die`, before this `die` Thm. 1 holds for the mutably borrowed pointer. Then, `die` copies cache value from mutably borrowed pointer to succeeding pointer, keeping the succeeding pointer cache in-sync with memory. ■

We now define $ObsState_{M1}$ for $\mathcal{M}_1$ as a tuple (P, R, M, S) with the pointer val field excluded from view. Let $\equiv$ be the equivalence relation between $\mathcal{M}_0$ and $\mathcal{M}_1$ defined as follows: $s_{M0}^{m_0} \equiv s_{M1}^{m_1} \leftrightarrow ObsState_{M0}(s_{M0}) = ObsState_{M1}(s_{M1})$. By Thm. 1, starting in equivalent observable states, both $\mathcal{M}_0$ and $\mathcal{M}_1$ operate in lock-step. Thus, the following theorem holds:

Theorem 2. *The relation $\equiv$ is both a forward and a backward simulation between $\mathcal{M}_0$ and $\mathcal{M}_1$.*

Thus, safety of $\mathcal{M}_1$ implies safety of $\mathcal{M}_0$ and vise versa.

III. VC GENERATION

```
1  fun main() {
2  BB0:
3    m00 = mem.init()
4    $m_{00}$
5    p0,m0 = mk_own 42, m00
6    $p_0.addr = 4 \wedge p_0.val = 42 \wedge$
7    $m_0 = m_{00}[p_0.addr \mapsto 42]$
8    q0 = mut_mkbor p0
9    p1 = mut_mksuc p0
10   $q_0.addr = p_0.addr \wedge q_0.val = p_0.val \wedge q_0.retval = x \wedge$
11   $p_1.addr = p_0.addr \wedge p_1.val = x \wedge p_1.retval = p_0.retval$
12   r1 = load q0, m0
13   $r_1 = q_0.val$
14   m1 = store r1 + 1, q0, m0
15   $q_1.addr = q_0.addr \wedge q_1.retval = q_0.retval \wedge$
16   $q_1.val = r_1 + 1 \wedge m_1 = m_0[q_1.addr \mapsto q_1.val]$
17   die q0
18   $q_1.val = q_1.retval$
19   r = load p1, m1
20   $r = p_1.val$
21   assert r == 43
22   $\neg(r = 43)$
23   halt
24 }
```

Fig. 3: Verification condition (VC) shown in yellow.

We introduce the general encoding of an OSEA-IR program and the modelling of mutable borrows in particular using the example in Fig. 3. Note that this example runs throughout this section. For now, we suggest the reader ignore the generated VC (in yellow). We focus on aliasing instructions and how the pointer cache is affected. The `mk_own` instruction defines `p0` writing `42` to both memory and the pointer cache maintaining *Cache Equivalence*. The `mut_mkbor`, `mut_mksuc` instructions create aliases `q0`, `p1` from `p0`. Here, the cache at `p0` is copied to `q0` and `p1`, again maintaining the cache equivalence invariant. The `q0` mutably borrowed alias updates memory (and its pointer cache) to `43`. It then surrenders access rights using the `die` instruction. At this point, the succeeding alias `p1` becomes active (top-of-(borrow)stack). However, for `p1` to maintain cache equivalence (Theorem 1), it must get a copy of `q0`'s cache. This is not straightforward since there is no explicit transfer instruction from `q0` to `p1`. The standard solution is to use shared memory so that `q0` can write to this memory on a `die` and the succeeding pointer `p1` can then read from this memory on next access. However, the aim of caching is to eschew memory accesses as much as possible to keep the operation (and VC) simple. The concrete semantics of $\mathcal{M}_1$ provides one alternative to accessing memory. There, a `die` instruction finds the succeeding pointer tag in the borrow store S_B and then searches through the register map R to update the pointer cache with the same tag. This mechanism is as (or more) expensive to execute symbolically as shared memory. An elegant solution proposed in RustHorn [3] uses a *prophecy variable* [9] to model the return of a mutable borrow in the VC. We adapt the scheme to VC generation (VCGen) for OSEA-IR. We now explain VCGen, emphasizing the role of prophecy variables to model return of a mutable borrow.

The VC is generated using the sym translation function. It builds up the VC in a recursive, bottom-up fashion on the abstract syntax tree of an OSEA-IR program. For simplicity of presentation, we assume that two fundamental sorts are used in the encoding: bit-vector of 64 bits, $bv(64)$, and a map between bit-vectors, $bv(64) \rightarrow bv(64)$. We now revisit the example and explain the VC for each line of source code. Line 4 models `mem.init` as m_{00}, a free variable. Line 6 models the `mk_own` instruction. It updates memory at $m_{00}[addr]$ to 42 and defines the fat pointer p_0. A fat pointer is modelled as a tuple $(addr, val, retval)$. Here $addr$ holds the address, val holds the current cache value (42 here), and $retval$ holds a prophecy value, the use of which will be laid out soon. A mutable borrow operation occurs in lines 10–11. The lender pointer p_0 creates two aliases, the mutable borrow q_0 and the succeeding pointer p_1. The location $p_0.addr$ is copied to both $q_0.addr$ and $p_1.addr$. The cache at $p_0.val$ is copied to $q_0.val$. To set up the return of the cache value from the mutably borrowed alias to the succeeding pointer, we *entangle* the $q_0.retval$ and $p_1.val$ field using a fresh prophecy value x. This prophecy x will resolve to the correct cache value when `q0` dies. When this happens, p_1 instantly gets the same value in its cache in $p_1.val$. Moving ahead, lines 13–16 model the increment of the value pointed to by q_0. Note that apart from updating the value in memory, the $q_0.val$ variant $q_1.val$ also gets the updated value. Finally, in line 18, the `die` operation causes the prophecy x to be constrained by equating $q_1.val$ and $q_1.retval$. As expected, this defines $p_1.val$ to get the correct cache value 43 maintaining cache equivalence. The transfer of cache from `q0` to `p1` is, therefore, modelled without any expensive symbolic operations involving memory accesses or register map lookups. In the end, we see that the generated VC is unsatisfiable and the property is valid.

We now describe the function sym for selected pointer operations. The semantics of `mk_own` is given in Fig. 4. We assume that an address ℓ is given by an external allocator. The allocator should follow the usual property that ℓ has not been allocated previously. Note that $p_0.retval$ field is free since an owning pointer does not return the cache value to another alias. We define sym for mutable borrow and die operations in Fig. 5. The mutable borrow aliasing operation copies the $addr$ field from the lender to the borrower and succeeding pointer. The cache is wired as follows. First, the mutably borrowing pointer gets the lender cache using $q_0.val = p_0.val$. Second, we entangle $p_1.val$ with the free symbol $q_0.retval$ using the $tngle$ macro. The macro itself entangles the first argument with the second by equating them. Third, $p_1.retval$ gets the prophecy in $p_0.retval$ to model cascading borrows (reborrows). The sym for *die* equates the given pointer's val and $retval$ field, constraining the prophecy value in $q.retval$ and returning the borrow.

In summary, the fat pointer concept is our workhorse in mapping two previous high level VCGen schemes to a low-level verification setting. First, the reference elimination mechanism is replaced by fat pointers that cache values. Second, a fat pointer field holds a prophecy value that expresses the cache value after returning from a mutable borrow.

$$sym(\texttt{p0, m1 = mk\_own n, m0}) \triangleq \exists \ell . (m_1 = m_0[\ell \mapsto n]) \land (p_0.addr = \ell) \land (p_0.val = n)$$

Fig. 4: Definition of *sym* for `mk_own`.

$$tngle(r_1, r_2) \triangleq r_1 = r_2$$

$$sym(\texttt{q\_0 = mut\_mkbor p\_0; p\_1 = mut\_mksuc p\_0}) \triangleq q_0.addr = p_0.addr \land q_0.val = p_0.val \land tngle(p_1.val, q_0.retval) \land p_1.retval = p_0.retval$$

$$sym(\texttt{die q}) \triangleq q.val = q.retval$$

Fig. 5: Definition of *sym* for mutable borrow, die, and *tngle* macro for entanglement.

IV. TOWARDS A PRACTICAL MACHINE

In Section II, we described $\mathcal{M}_0$ and $\mathcal{M}_1$, both machines that could only allocate a single word through `mk_own`. We lift this restriction now in $\mathcal{M}_2$. To allocate multiple words (wide allocations), we change the `mk_own` syntax. Instead of taking a bit-vector to write to memory, it now takes a bit-vector *allocation size* argument. For cache equivalence to hold, the pointer cache width must now be wide enough to cache multi-byte allocation data. This complicates the design of the cache. To keep things simple, instead of hard-wiring the pointer cache to replicate memory contents, we only cache a *summary* of the data in memory and provide operations to set and get the cache value using `set_cache` and `get_cache`, respectively. A property to be verified can be cached at the pointer. Pointer aliasing operations copy the value as before. The decoupling of cache from `load` and `store` operations does introduce burden on the programmer to update the cache as required. As we move towards a practical machine, we also add a new unique (u) variant to pointer type *ptype*. A unique pointer is created using `begin_unique` and `end_unique` instructions.

The syntax of these new instructions is given in Fig. 6. The `mk_own` instruction takes three arguments - the bit-vector to write, the size (in bytes) of the allocation and the incoming memory to update. The operation now does not update memory or the pointer cache since that is the programmer's responsibility. The `begin_unique` and `end_unique` operations take a copied (unique) pointer and define a unique (copied) pointer with the same *addr* and *val* fields as the source pointer. These operations are useful when the user only wants to mark a pointer as unique temporarily. The `get_cache` instruction returns the *val* field of a pointer. the `set_cache` instruction takes a pointer and a value. It then defines a new pointer where all fields are the same as the source pointer, except the *val* field that has been updated to the given value.

Verification pipeline. To evaluate the efficacy of ownership intrinsics for verification, we use the SEABMC bit-precise bounded model checker. SEABMC operates on LLVM IR programs. For this work, the SEABMC VCGen process has been enhanced to handle ownership instructions. It is cumbersome to construct low-level OSEA-IR programs by hand to be verified in SEABMC. To ease the task, we provide an API for adding ownership semantics to C programs resulting in a C-like programming language with ownership semantics. The API is in the form of C macros. The C program is compiled to an OSEA-IR program. The low-level OSEA-IR program then generates the VC in SMT-LIB form. This is finally sent to an SMT solver. We discuss the API using the example high level program in Fig. 7. The program starts in line 3, the `MK_OWN` macro allocates a byte of memory to an owning pointer. The next line uses the `nd_char` function to assign a non-deterministic char to `c`. The value of `c` is constrained to be `42` using an **`assume`** statement. In line 6, the cache at pointer `p` is set to the value of `c` using the `SET_CACHE` macro. The value is also stored in memory using pointer `p`. The macro `MUT_BORROW` in line 9 then creates a mutable borrow. Internally, the macro expands to `mut_mkbor` and `mut_mksuc` with `b` getting the mutable borrow and `p` getting the succeeding pointer. Next, the non-deterministic boolean value from `nd_bool` is used in line 10 to conditionally update `b`'s cache to a non-deterministic value greater than `43`. The `escapeToMemory` function takes the address of `b` thwarting any optimisation attempts to promote `b` to a register. Finally, `b` dies in line 17 using the macro `DIE`. The succeeding pointer's cache is now read using `GET_CACHE` into `r` in line 19. The **`sassert`** (static assert) then checks that the value of `r` is either `42` or greater than `43`.

$$\begin{array}{lll}\langle \mathrm{RDEF} \rangle & ::= & \ldots \mid \langle \mathrm{P} \rangle, \langle \mathrm{M} \rangle = \texttt{mk\_own}\ \langle \mathrm{R} \rangle, \langle \mathrm{M} \rangle \mid \\ & & \langle \mathrm{P} \rangle = \texttt{begin\_unique}\ \langle \mathrm{P} \rangle \mid \langle \mathrm{P} \rangle = \texttt{end\_unique}\ \langle \mathrm{P} \rangle \mid \\ & & \langle \mathrm{P} \rangle = \texttt{set\_cache}\ \langle \mathrm{P} \rangle, \langle \mathrm{R} \rangle \mid \langle \mathrm{R} \rangle = \texttt{get\_cache}\ \langle \mathrm{P} \rangle \end{array}$$

Fig. 6: Grammar of new instructions for OSEA-IR.

```
1  extern void escapeToMemory(char *);
2  int main() {
3    char *p = MK_OWN(0, sizeof(char));
4    char c = nd_char();
5    assume (c == 42);
6    SET_CACHE(p, c);
7    *p = c;
8    char *b;
9    MUT_BORROW(b, p);
10   if (nd_bool()) {
11     c = nd_char();
12     assume(c > 43);
13     SET_CACHE(b, c);
14     *b = c;
15     escapeToMemory(b);
16   }
17   DIE(b);
18   char r;
19   GET_CACHE(p, r);
20   sassert(r == 42 || r > 43);
21   return 0;}
```

Fig. 7: A C program with Ownership macros in yellow.

For the program in Fig. 7, Fig. 8a is its OSEA-IR form and Fig. 8b is the generated VC. We now describe the VCGen in $\mathcal{M}_2$ using Fig. 8. The ownership instructions are highlighted in yellow in both figures. The `MK_OWN` macro in C becomes the `mk_own` instruction in OSEA-IR and is translated to SMT-LIB form using *sym*. Note that in $\mathcal{M}_2$, `mk_own` does not write to memory or update the pointer cache. The symbolic semantics therefore only allocates memory and provides a previously unallocated address $addr_0$. The `set_cache` instruction in line 7 defines a pointer p_3 with the same *addr* as p_2 and the cache updated to r_5. The mutable borrow occurs in lines 9–10. The

```
 1 fun main() {
 2 BB0:
 3   m3 = mem.init()
 4   p2, m0 = mk_own 1, m3
 5   r5 = nd_char()
 6   r6 = r5 == 42
 7   p3 = set_cache p2 r5
 8   m1 = store r5, p3, m0
 9   p5 = mut_mkbor p3
10   p6 = mut_mksuc p3
11   r15 = nd_bool();
12   r17 = r15 == 42
13   br r17, ERR, BB1
14
15 BB1:
16   r18 = nd_char()
17   r19 = r18 > 43
18   r20 = r6 && r19
19   p23 = set_cache p5 r18
20   m2 = store r18, p23, m1
21   escapeToMemory(p0)
22   br ERR
23
24 ERR:
25   r22 = select r17, r6, r20
26   p24 = select r17, p5, p23
27   die p24
28   r29 = get_cache p6
29   r30 = r29 == 42
30   r31 = r29 > 43
31   r32 = r30 || r31
32   A = not r32
33   assume A
34   assert false
35   halt
36 }
```

(a) OSEA-IR program.

$p_2.addr = addr_0 \wedge m_0 = m_3 \wedge$
$r_6 = (r_5 = 0) \wedge$
$p_3.addr = p_2.addr \wedge p_3.val = r_5 \wedge$
$p_5.addr = p_3.addr \wedge p_6.addr = p_3.addr \wedge$
$tngle(p_5.retval, p_6.val) \wedge p_5.val = p_3.val \wedge$
$r_{17} = (r_{15} = 0) \wedge$
$r_{19} = r_{18} > 1 \wedge$
$r_{20} = r_6 \wedge r_{19} \wedge$
$p_{23}.addr = p_5.addr \wedge p_{23}.val = r_{18} \wedge$
$r_{22} = ite(r_{17}, r_6, r_{20}) \wedge$
$p_{24} = ite(r_{17}, p_5, p_{23}) \wedge$
$p_{24}.retval = p_{24}.val \wedge r_{22} \wedge$
$r_{29} = p_6.val \wedge$
$r_{30} = r_{29} = 0 \wedge$
$r_{31} = r_{29} > 1 \wedge$
$r_{32} = (r_{30} \vee r_{31}) \wedge$
$a = \neg r_{32} \wedge$
$a \wedge$
$\neg false$

(b) SMT-LIB program.

Fig. 8: Program from Fig. 7 in OSEA-IR and SMT-LIB forms. Ownership intrinsics and their counterpart expressions in SMT are highlighted in yellow.

```
enum status {O, C};
int unit_proof(const char **fnames,
    int n) {
  FILE *f[MAX];// assume n < MAX
  for(int i=0; i < n; i++) {
    set_shad(f[i], O);
    f[i] = open(fnames[i], "w");}
  size_t choose = nd_size_t();
  assume(choose < n);
  FILE *file = f[choose];
  write(file);
  // check file closed
  sassert(get_shad(file) == C);}
```

(a) A unit proof.

```
void write(FILE *fp) {
  // check file opened
  sassert(get_shad(fp) == O);
  fputc('a', fp);
  // mark closed
  set_shad(fp, C);
  fclose(fp);}
```

(b) An SUT.

Fig. 9: An example of typestate storage in shadow memory.

semantics copies the lender $p_3.addr$ to $p_5.addr$ and $p_6.addr$. The val and $retval$ fields are set up as usual. The cache of the borrowed pointer p_5 is conditionally updated in line 19. The borrowed (variant) pointer p_{24} dies in line 27 with the usual semantics. The cache of the succeeding pointer p_6 is read into r_{29} in line 28. The lines 29–32 set up verification such that if an execution satisfies **assume** `A` then it reaches the error state (`assert` **`false`**). An important consequence of ownership semantics is that the SMT-LIB program does not need to model the `store` instruction in line 20.

V. EVALUATION

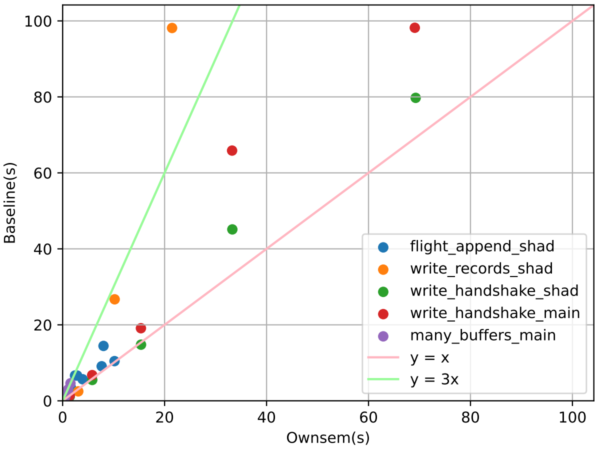

Fig. 10: Solve time (in sec.) using ownership semantics vs baseline.

We would like to cache verification properties for practical programs. We first describe properties of interest and our baseline property carrying mechanisms through the example in Fig. 9. Here, we want to check that a `write` function in Fig. 9b writes only to an *open* file and the file is always *closed* after a write. The state of the file object is encoded as a *typestate* property [13] that records (and checks) operations that have occurred on an object. The `write` function is called using the *unit proof* harness in Fig. 9a. It defines `n` file pointers based on user input. The typestate is marked open for each file pointer in *shadow memory* using the `set_shad` function. Shadow memory is an address map ($addr$ to $bv(64)$). It is used to stash verification relevant object metadata. In practice, shadow memory is provided by verification and testing tools like SEABMC and Memcheck [14]. If shadow memory is not available then metadata can be stashed in a separate main (program) memory allocation, which is available when the program is built in debug or verification mode. In the example, after the typestate is set to open for each file pointer, we choose one file pointer `file` out of `n` for calling `write` — the system under test (SUT). In the SUT, we first check that the typestate of the file object is open using `get_shad` to get the typestate value. Then the char `'a'` is written and the file is closed, with the typestate marked as closed. Finally, the harness checks that the file typestate is indeed closed.

Note in the given unit proof, the write and read of shadow memory can resolve to `1`-of-`n` allocations since any file object can be chosen. Therefore, in the VC, memory access involves solving an ITE (if-then-else) expression for a choice. However, solving this ITE is redundant since we only want to check that a given operation occurred on the chosen file pointer. An alternative would be to store typestate in the file (fat) pointer cache itself. With this optimisation, an ITE would not be traversed since read, writes of the typestate would be at the pointer (using `get_cache` and `set_cache`) leading to simpler VC.

We base our experiments on this idea utilizing the SEABMC model checking engine for SEAHORN. SEABMC originally takes a SEA-IR program as input and generates VC that are solved by an SMT solver. We enhance SEABMC to now take OSEA-IR programs as input. Using the C macro API, C unit proofs are compiled to SMT. We then measure how typestate cached at pointers compares to typestate stored in memory.

The C unit proofs we work with come from `mbedTLS` [15], a C library of cryptographic primitives, SSL/TLS and DTLS protocols. In particular, we look at three functions in `ssl_msg.c` that handles SSL message construction and de-construction. The flow we consider are (1) `flight_append` that appends messages to the current flight of messages, (2) `write_records` that encrypts messages into records and sends them on the wire, and, (3) `write_handshake` that writes handshake messages. Each SUT operates on a byte buffer data structure. We are interested in recording and checking typestate properties for such a buffer. However, similar to example Fig. 9, the unit proof is set up such that a single byte buffer pointer may point to 1-of-n buffer objects. Therefore, we study if using pointer caching improves solver performance.

The experiments are run on an Intel(R) Xeon(R) E5-2680 CPU operating at 2.70GHz with 64 GiB of main memory. The generated VC are solved using Z3 [16] `smtfd` tactic. The scatter plot in Fig. 10 shows the solving time for unit proof with ownership semantics (ownsem) in the x-axis. The y-axis records the solving time for the same unit proof that either uses shadow memory or main memory as the baseline. The legend clarifies the memory we compare against using either *shad* or *main* in the name suffix. We run each flow for increasing number of byte buffers behind a pointer (e.g., 2, 4, 6, ...) and stop when the running time in either ownsem or baseline mode reaches 100 seconds. The `many_buffers` benchmark is hand-crafted and shows a consistent 3x improvement for ownsem. The flows from `mbedTLS` show more spread. For small number of buffers, ownsem and baseline are usually head-to-head. As the number of buffers increase, ownsem outperforms baseline. For `write_handshake_shad`, the performance boost is 1.3x when using 8 buffers. For `write_records_shad`, the performance boost at 8 buffers is 5x. This is shown on the scatter plot. Looking at the SMT solver metrics, we see internal metrics like `sat conflicts` and `sat backjumps` correlate with the timings (See [11] for details). When a unit proof is faster, fewer conflicts and backjumps are seen compared to the baseline. This is indirect evidence that the performance boost is due to VC simplicity.

Simplification of VC itself depends on how many memory accesses can be soundly replaced by pointer cache accesses. VC simplicity is affected by (1) the extent a conditional typestate check depends on program memory state, and (2) the number of typestate memory accesses as a fraction of the total number of memory accesses. As an example, for a conditional check such as `if(*unrelated_ptr == 1){get_cache(ptr);}`, a read of `ptr` cache using `get_cache` does not access `ptr` memory. However, for the check to be reachable, the guarding `if` condition does require a memory access. Therefore, it is not always possible to remove dependency on program memory for conditional typestate checks. Also if the unit proof (and the SUT) do not set/get typestate checks frequently then replacing such checks by pointer cache accesses has limited benefits. The data and units proofs to reproduce our experiments are available at https://github.com/priyasiddharth/mbedtls-ownsem.

Overall, a speedup in solving time occurs as expected. The speedup is due to simpler VC. However, the speedup is sensitive to the property expressed as a typestate check and number of operations on object (pointer) that affect typestate.

VI. RELATED WORK

RustBelt [17], Oxide [18] formalize subsets of high level Rust. RustBelt uses a continuation passing style functional language to describe the semantics. Oxide uses a high level language similar to Rust. These approaches do not map directly to a low–level register machine like LLVM. Stacked Borrows [12] formulates Rust ownership semantics as a stack discipline working on de–sugared (MIR) Rust syntax that represents memory by an address map. Its aim is to provide a reference semantics for the borrow checker separate from the production version in the Rust compiler. Stacked Borrows is implemented in the MIR interpreter (MIRI) and is part of the Rust standard distribution. We rely heavily on stacked borrows to design low-level semantics for this paper.

The Move Prover [5] uses reference elimination to replace a reference by its data. It assumes an alias free memory model and solves the problem of return of a mutable borrow by recording the origin (lender) of a mutable borrow and returning data to it explicitly rather than utilizing prophecies. RustHorn [3] uses a prophecy value to model return of a mutable borrow and assumes a safe Rust-like language and, therefore, forgoes modelling an address map entirely. RustHornBelt [6] extends this work to cover unsafe Rust where the safety in the unsafe part is manually proven in Iris [19], a concurrent separation logic prover built on top of Coq [20].

Verus [21], Prusti [22], and Creusot [23] are deductive verifers for Rust. Creusot uses RustHorn style prophecy variables. These deductive tools can model complicated features of the language, like polymorphism, directly. This paper focuses on low-level memory manipulating programs.

The memory models used in CBMC [24], LLBMC [25], and stock SeaBMC [10] assume an unsafe language allowing unrestricted aliasing of pointers and support pointer arithmetic. Kani [26] is a Rust verifier that compiles to goto-cc, the same low-level backend as CBMC. Ownership information, though, is lost is this conversion. Overall, we expect these low-level tools would perform similar to our baseline experiments.

VII. CONCLUSION

We describe formal ownership semantics for multiple low-level machines of increasing complexity. Particularly, we explain the mechanism for caching values at fat pointers and keeping the values in-sync with memory. We use the given semantics to describe VCGen for BMC such that the number of occurrences of memory accesses in the VC is reduced. For this we model return of mutable borrows using prophecy values added to fat pointers. We evaluate the efficiency of generated VC by experiments using the SEABMC tool. Overall, we see improvements in solving time and attribute it to the simplicity of VC.

REFERENCES

[1] M. Fähndrich and R. DeLine, "Adoption and focus: Practical linear types for imperative programming," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds. ACM, 2002, pp. 13–24. [Online]. Available: https://doi.org/10.1145/512529.512532

[2] D. Grossman, J. G. Morrisett, T. Jim, M. W. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in cyclone," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds. ACM, 2002, pp. 282–293. [Online]. Available: https://doi.org/10.1145/512529.512563

[3] Y. Matsushita, T. Tsukada, and N. Kobayashi, "Rusthorn: Chc-based verification for rust programs," *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 4, pp. 15:1–15:54, 2021. [Online]. Available: https://doi.org/10.1145/3462205

[4] J. Noble, J. Vitek, and J. Potter, "Flexible alias protection," in *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, ser. Lecture Notes in Computer Science, E. Jul, Ed., vol. 1445. Springer, 1998, pp. 158–185. [Online]. Available: https://doi.org/10.1007/BFb0054091

[5] D. L. Dill, W. Grieskamp, J. Park, S. Qadeer, M. Xu, and J. E. Zhong, "Fast and reliable formal verification of smart contracts with the move prover," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 183–200. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_10

[6] Y. Matsushita, X. Denis, J. Jourdan, and D. Dreyer, "Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code," in *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, R. Jhala and I. Dillig, Eds. ACM, 2022, pp. 841–856. [Online]. Available: https://doi.org/10.1145/3519939.3523704

[7] L. Developers. (2024) llvm website. [Online]. Available: https://llvm.org/docs/LangRef.html#noalias-and-alias-scope-metadata

[8] R. Developers. (2024) Rust website. [Online]. Available: https://github.com/rust-lang/rust/issues/54878#issuecomment-429578187

[9] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991. [Online]. Available: https://doi.org/10.1016/0304-3975(91)90224-P

[10] S. Priya, Y. Su, Y. Bao, X. Zhou, Y. Vizel, and A. Gurfinkel, "Bounded model checking for LLVM," in *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, A. Griggio and N. Rungta, Eds. IEEE, 2022, pp. 214–224. [Online]. Available: https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_28

[11] S. Priya and A. Gurfinkel, "Ownership in low-level intermediate representation," 2024. [Online]. Available: https://arxiv.org/abs/2408.04043

[12] R. Jung, H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: an aliasing model for rust," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 41:1–41:32, 2020. [Online]. Available: https://doi.org/10.1145/3371109

[13] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 157–171, 1986.

[14] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, San Diego, California, USA, June 13-15, 2007*, C. Krintz, S. Hand, and D. Tarditi, Eds. ACM, 2007, pp. 65–74. [Online]. Available: https://doi.org/10.1145/1254810.1254820

[15] mbedTLS Developers. (2023) mbedtls project. [Online]. Available: https://github.com/Mbed-TLS/mbedtls

[16] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[17] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 66:1–66:34, 2018. [Online]. Available: https://doi.org/10.1145/3158154

[18] A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed, "Oxide: The essence of rust," *CoRR*, vol. abs/1903.00982, 2019. [Online]. Available: http://arxiv.org/abs/1903.00982

[19] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, S. K. Rajamani and D. Walker, Eds. ACM, 2015, pp. 637–650. [Online]. Available: https://doi.org/10.1145/2676726.2676980

[20] C. Developers. (2024) The coq proof assistant. [Online]. Available: https://coq.inria.fr/

[21] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 286–315, 2023. [Online]. Available: https://doi.org/10.1145/3586037

[22] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust," in *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, ser. Lecture Notes in Computer Science, J. V. Deshmukh, K. Havelund, and I. Perez, Eds., vol. 13260. Springer, 2022, pp. 88–108. [Online]. Available: https://doi.org/10.1007/978-3-031-06773-0_5

[23] X. Denis, J. Jourdan, and C. Marché, "Creusot: A foundry for the deductive verification of rust programs," in *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*, ser. Lecture Notes in Computer Science, A. Riesco and M. Zhang, Eds., vol. 13478. Springer, 2022, pp. 90–105. [Online]. Available: https://doi.org/10.1007/978-3-031-17244-1_6

[24] D. Kroening, P. Schrammel, and M. Tautschnig, "CBMC: the C bounded model checker," *CoRR*, vol. abs/2302.02384, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.02384

[25] F. Merz, S. Falke, and C. Sinz, "LLBMC: bounded model checking of C and C++ programs using a compiler IR," in *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Joshi, P. Müller, and A. Podelski, Eds., vol. 7152. Springer, 2012, pp. 146–161.

[26] K. Developers. (2023) The kani book. [Online]. Available: https://model-checking.github.io/kani/