A Formal Y86 Simulator with CHERI Features

Carl Kwan, Yutong Xin, William D. Young
The University of Texas at Austin, Austin, TX, USA
{carlkwan,maxxin,byoung}@cs.utexas.edu

***Abstract*—We present a formal executable model of CHERI architectural features integrated with a formalized Y86 ISA. CHERI is an extension of conventional hardware ISAs centralized around capabilities, which are descriptions of permissions at the hardware level that can be used in place of pointers. CHERI enables fine-grained memory protection and highly scalable software compartmentalization to mitigate security vulnerabilities beyond what can be done by current architectures. We use our formal model to prove the memory protection and security features of CHERI itself, prototype extensions to CHERI, and verify the correctness of machine code involving capabilities. Since it is executable, our model also serves as a *symbolic simulator* of a CHERI augmented x86-like processor, allowing step-by-step validation of CHERI capability features in a controlled, formal environment.**

We are motivated by the adoption of CHERI by industry leaders that design real-world hardware, the need for industrial-strength tools to perform formal CHERI analyses, and the current development of CHERI extensions to x86 platforms. We build our model in an all-in-one first-order logic and programming system. Our model enables formal verification of CHERI designs and the rapid-prototyping of new CHERI architectural features, and we apply existing industry push-button verification tools and custom heuristics to prove the correctness of CHERI artifacts.

***Index Terms*—Formal models, operational semantics, theorem proving, ACL2, capabilities, Y86**

I. Introduction

Capability Hardware Enhanced RISC Instructions (CHERI) is a set of architectural features that enables software compartmentalization and more precise and flexible memory protection than traditional memory protection mechanisms [1]. It is designed to mitigate various types of security vulnerabilities, particularly those related to memory safety, via various hardening techniques. Recently, Arm has shipped Morello, a multicore, superscalar, Neoverse N1-based processor, System-on-Chip, and prototype board to academic and industry security experts [2]. These shipped Arm hardware artifacts are fully integrated with CHERI protection features. Microsoft has described CHERI as a "building block for higher-level security abstractions" [3].

At the center of CHERI are capabilities, which are unforgeable tokens in hardware. These tokens describe the range and actions permitted for various kinds of memory accesses. Capabilities provide a way to control and restrict access to memory at a fine-grained level by including information about the base address of the memory range for which access is permitted, the size of the range, and the permissions associated with that range. One can view capabilities as *fat-pointers*, i.e. pointers containing metadata associated with the memory address to which the pointer value points. In the case of a capability, the metadata are the permissions, memory bounds, and the capability's object type.

In CHERI systems, *all* memory access is controlled by capabilities. This fundamentally changes how machine code programs are executed, requires additions to instruction set architectures, and modifies architectural state parameters, such as registers, program counters, etc. The CHERI ISA specification technical reports document each of these additional instructions and other architectural contributions and changes. However, the most recent CHERI ISA contains over 500 pages of technical details, motivation, design specifications, algorithms, etc [4].

We present a formal executable model of the Y86 ISA integrated with CHERI, which we call CHERI-Y86. Y86 is an ISA that executes a simplified subset of the x86-64 instruction set [5]. It was originally introduced by Randal Bryant and David O'Hallaron, and is commonly used by hundreds of institutions to teach computer systems.

It is important that CHERI-Y86 faithfully implements the CHERI-ISA specification [4]: every opcode encoding, register-state extension, permission check, and capability-compression routine is lifted directly from the official manual. To help prevent transcription or interpretation errors, we validated our model using an adapted CHERI conformance test suite designed based on the CHERI-x86-64 ISA descriptions, ensuring correct behavior for all non-deprecated capability-aware instructions. We apply our model primarily to the verification of machine code programs that involve capabilities and formally prove the correctness of CHERI architectural design and security features. This includes proofs for all non-deprecated capability-aware instructions, verification of CHERI's capability-compression scheme, guarded accesses to any state parameter in CHERI-Y86, and explicit formal executable specifications for CHERI methods and algorithms. CHERI-Y86 is publicly available[1].

There are two additional broader motivations behind our development of CHERI-Y86. First, Y86 is similar to the x86 ISA. While CHERI specifications for x86 are not yet fully elaborated, CHERI-Y86 serves as a useful proof-of-concept for building future models and hardware that integrate

[1]https://gitlab.com/cheri-acl2/y86-cheri-capabilities

https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_22

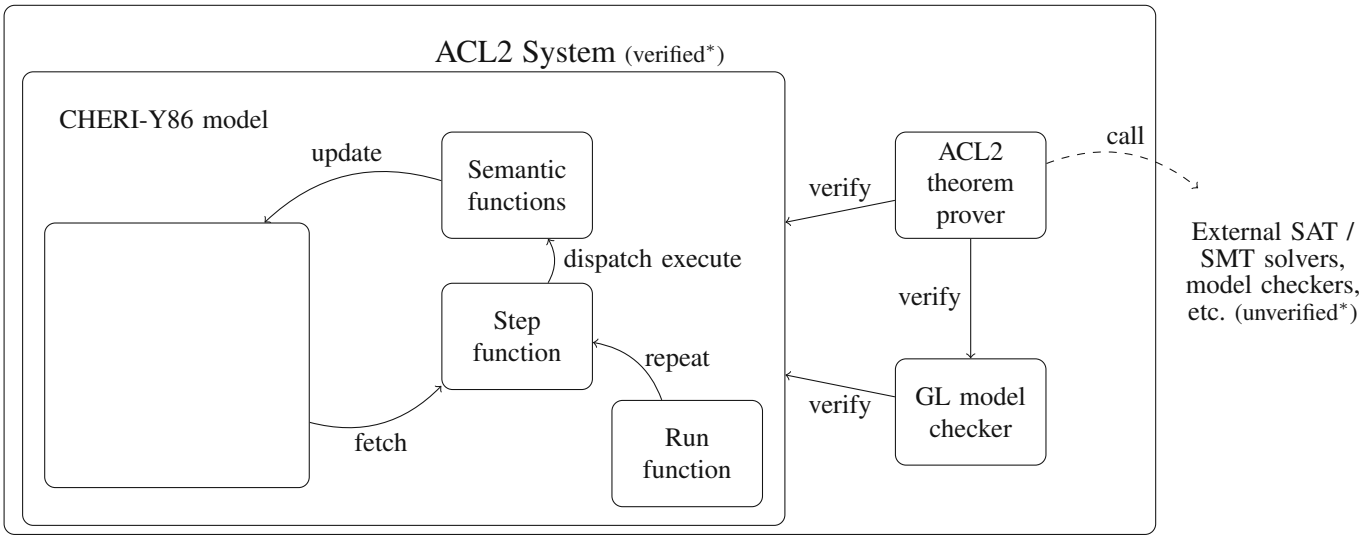

Fig. 1. ACL2 CHERI-Y86 model overview

CHERI with x86. We developed the CHERI-Y86 features in a purposely modular fashion. This is to enable future integration of our model with formal models of the x86 ISA (pending completion of the CHERI x86 specification). This will be invaluable to industry and academic efforts to build highly reliable and secure x86 hardware and low-level software systems. Second, Y86 is much simpler than x86. One of our major goals is to elaborate key CHERI features with our model rather than address a lot of specific architectural details. By using a stripped-down ISA, we emphasize the already numerous architectural features of CHERI without introducing the added burden of an otherwise complicated instruction set. This makes CHERI accessible to students and professionals, and enables the introduction of CHERI features to engineers of other architectures that wish to secure their designs without having to learn the complexities of another platform.

Our model is built in ACL2 [6], a first-order logic and proof system that is fully programmable, extensible, and fast, with support for industry-strength push-button verification tools. Employing ACL2 in building our model of CHERI-Y86 combines the convenience of fully automatic tools and a general purpose theorem prover, enabling users to offload tedious brute-force verification processes and focus more on purpose-driven model and proof design aspects.

II. Related Work

A *capability* is an unforgeable token of authority to access a system object, along with a set of operations that the holder of the capability is permitted to perform on the object. Possession of a capability is sufficient to guarantee that the holder has the indicated access permissions to the object. In theory, the use of capabilities obviates the need for access control lists or similar security structures; entities are provided exactly those capabilities needed to perform their function. This potentially facilitates fine grained access control and enforcement of least privilege, but also necessitates that the system prevent user programs from forging or modifying capabilities. Such protections may be implemented in hardware or software.

Although several earlier systems included capability-like addressing mechanisms, the term "capability" was first introduced by Dennis and Van Horn [7] in the mid-1960s as part of an hypothetical operating system supervisor for a multiprogramming system. Capabilities addressed a variety of issues in OS security, including sharing and cooperation among processes, protection of processes, and naming of objects [8, p. 41]. They subsequently were adopted by a number of hardware and software systems. Levy's *Capability-Based Computer Systems* [8] provides a detailed overview of capability-oriented computer system design through the 1980's. More recently, capabilities have seen adoption in a variety of systems. The CHERI documentation provides an excellent overview [9]. Some notable recent systems implementing capability-based security include: EROS [10] (now CapROS and Coyotos [11]), a capability system that permits creating confined subsystems; and seL4 [12], a formally verified, capability-oriented microkernel.

Formal verification, specifically machine checked proofs of security properties, is especially desirable in the case of capability-based systems in light of claims by Boebert [13] and others that unmodified capability systems cannot enforce fundamental access control properties. Both EROS and seL4 included significant verification efforts. A proof [14] of the EROS confinement properties was carried out by the developers, though it appears that the proof is not machine checked. The seL4 microkernel has been subjected to formal analyses beyond any other comparable system. Extensive proofs [15] of seL4's access control system and of the functional correctness of the kernel have been carried out using Isabelle/HOL.

Lightweight formal methods have also been deployed to CHERI in particular. CHERI-MIPS and CHERI-RISC-V benefit from a suite of tools that enable their formal specification

in a domain-specific language called Sail [16]. The advantage to this approach is that generating Isabelle specifications from Sail is automatic, and an impressive set of security properties for CHERI-MIPS has been verified in Isabelle [17]. Extensive formal verification efforts behind the Morello architecture also deploy an extensive suite of translation layers, validation suites / test generators, specification languages, and various theorem proving tools (mostly revolving around Isabelle) to ensure vital properties involving capability monotonicity [18]. Similarly, encoding / decoding aspects of CHERI Concentrate have also been verified using Sail's SMT backend in the CHERIoT work [19]. However, proving properties about these specifications require a translation layer from Sail or other specification languages to the Isabelle theorem prover. Interactions between different systems introduces logical concerns involving soundness. Moreover, while Isabelle can express a wide breadth of mathematics, support for execution and simulation is limited. The work we present in this paper enables the efficient modelling, simulation, and verification of CHERI features for an x86-like ISA all in a single logic.

A subset of the CHERI ISA has also been formalized in the operational semantics approach using Coq [20], [21]. The focus of this work is to verify untrusted machine code programs do not violate certain security properties on a CPU which supports capabilities. We are also interested in verifying machine code programs using our simulator, but our focus with this work is to develop a formal model with formalized CHERI features that gets us closer to existing real-world ISAs.

Formal proofs have become an essential component of industrial hardware development, from the proof of the AMD floating point multiplication unit [22]. All major hardware manufacturers utilize automated reasoning tools to deal with the immense complexity of modern digital circuitry [23].

ACL2 [6], specifically, has been used to prove properties of digital systems, both hardware and software, for decades and is in widespread use in industry [23]. Numerous machine models have been built in ACL2 over the years including the Sun Java Virtual Machine, the Motorola CAP digital processor, and many more. The ACL2 homepage contains an extensive publications list.[2] Most relevant to the current project are those modelling the x86 ISA [24]–[27]. The ACL2 x86 model is similar to CHERI-Y86 in that they both simulate the behaviour of a machine by way of operational semantic functions which update a state object formalized as an ACL2 single-threaded object. The effect of machine code on both models can be symbolically simulated using Boyer-Moore clock functions and formally verified. An advantage to the ACL2 x86 model is that it is the largest and most comprehensive specification of an x86 machine, to the point where it can be used to boot Alpine Linux. In fact, the Sail specification of the x86 architecture is mechanically translated from the ACL2 specification.

III. Formalized CHERI Features

The underlying CHERI protection feature at the level in which we are interested is the restriction of access to architectural components (e.g. memory, registers, or code objects) by way of minimizing the permissions available to other components (e.g. executable instructions), thus compartmentalizing and limiting the effects of bugs or other vulnerabilities. Permissions and the scope of permissions are controlled by *capabilities*.

The CHERI ISA specifies two representations of capabilities and we formalize both. *CHERI Concentrate* is the explicit format for the representation of capabilities *in memory*. Capabilities represented abstractly with software-accessible fields (some of which may not be explicitly stored in memory) are called *architectural capabilities*. As part of our formalization, we develop and verify ACL2 functions that allow a user to easily convert between architectural and memory-resident capabilities. The proofs of these conversion functions amount to verifying that converting from architectural capabilities to memory-resident capabilities and back again recovers the information that was originally in the architectural capabilities, and vice-versa. In this section, we describe our formalizations of CHERI architectural capabilities, CHERI Concentrate, and protection features.

A. CHERI Concentrate

System bandwidth, throughput, and memory limitations have led to many iterative modifications to the compressed format of capabilities, requiring the need for encode and decode functions. CHERI uses a compressed format called CHERI Concentrate, which specifies a 64-bit representation for capabilities on 32-bit systems and a 128-bit representation for capabilities on 64-bit systems. We are only interested in 128-bit capabilities, the format is summarized by Figure 2. The format contains:

- a: a 64-bit address;
- p: 16 permission bits including various read, write, and execute permissions;
- o: 18 object type (otype) bits indicating whether and how a capability is sealed;
- I: 1 internal exponent bit determining whether the bounds are in exponent format;
- $t_{[11:3]}$ & t_E: 12 bits for computing the capability's "top" 14-bit memory address;
- $b_{[13:3]}$ & b_E: 14 bits for computing the capability's "base" 14-bit memory address.

CHERI Concentrate uses a compressed floating point representation to encode the "top" t and "base" b bounds relative to a capability's address. It supports two formats: if I is set, then the lowest three bits b_E and t_E of b and t, respectively, are used to represent an exponent at the expense of three bits of precision; otherwise, the exponent is considered to be zero giving back b and t their lower three bits. An interested reader should consult the CHERI ISA [9] and original CHERI Concentrate [28] paper for details.

In addition to formalizing CHERI Concentrate, we also present the formal verification of the encode / decode function properties. The role of the encode / decode functions is to compress / decompress the bounds for the region of memory

[2]https://www.cs.utexas.edu/users/moore/publications/acl2-papers.html.

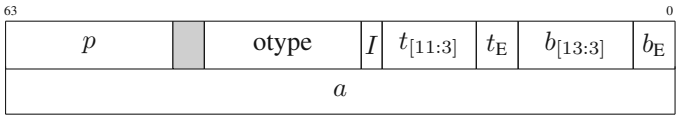

Fig. 2. CHERI Concentrate 128-bit format.

a capability is permitted to access, which are specified by a base address and a length. When added together, the base address and the length form a top address, i.e. the upper bound in memory a capability is permitted to access. Due to compression, some accuracy is lost after encoding the top and base, except in certain circumstances. Let b_0, ℓ_0, and t_0 be the base, length, and top before encoding, respectively. Let b_1, ℓ_1, and t_1 be the result of encoding and then decoding b_0, ℓ_0, and t_0, respectively. These are the properties we verify:

1) $b_0 \geq b_1$ for any b_0, t_0, and address;
2) $b_0 - b_1 \leq 2^{E+3}$ for any b_0, t_0, and address;
3) $t_0 \leq t_1$ for any b_0, t_0, and address;
4) $t_1 - t_0 \leq 2^{E+3}$ for any b_0, t_0, and address;
5) $b_0 = b_1$ and $t_0 = t_1$ when the lower $E + 3$ bits of b_0 and t_0 are zero;
6) $b_0 = b_1$ and $t_0 = t_1$ when $\ell_0 < 2^{12}$.

Here E is a 6-bit value representing an exponent when $\ell_0 \geq 2^{12}$. Properties (1) and (3) ensure that compression does not relax the memory bounds originally intended by the capability, thus ensuring secure memory compartmentalization. Properties (2) and (4) guarantee that any rounding error resulting from compression is not too large. Properties (5) and (6) specify the exact conditions for which the bounds can be recovered exactly. We discuss the verification of these properties and the formalization of CHERI Concentrate in Section VI.

B. Architectural Capabilities

CHERI Concentrate is a format for storing capabilities *in memory*. We make a distinction between *architectural capabilities* and capabilities in memory. Architectural capabilities contain software-accessible fields that are not reflected explicitly by, but can still be inferred from, a given capability's in-memory representation. In our model, architectural capabilities contain the following fields: 1-bit validity tag; 16-bit permissions (perms); 18-bit object type (otype); 64-bit offset; 64-bit base; and 64-bit length. The base is the lower bound of the memory region that can be dereferenced by a capability. Adding length to base gives the upper bound (a.k.a. top) of the memory region that can be dereferenced by a capability. Adding offset to base gives the address when the capability is used as a pointer.

System constraints necessitate compressing the information provided by an architectural capability into the format described by CHERI Concentrate, which is then stored in memory. We formalize a kind of program interface in which numerous functions translate between the human readable architectural capabilities format and the compressed raw bits of CHERI Concentrate. For fields such as perms or otype, simple direct bit-to-integer and integer-to-bit translations suffice. Translating offset, base, and length fields from an architectural capability to the base, top, and address fields in CHERI Concentrate requires much more sophisticated interface functions and must satisfy the properties discussed in Section III-A

Name	Field	Description
Registers	`rgf`	16 general-purpose 128-bit registers
RIP	`rip`	Program counter
ZF	`zf`	Zero flag
SF	`sf`	Sign flag
OF	`of`	Overflow flag
Memory	`mem`	2^{64} bytes modelled with 2^{24} addresses
MS	`ms`	Model state, indicates model errors

TABLE I
CHERI-Y86 MACHINE STATE OBJECT

IV. THE ACL2 CHERI-Y86 MODEL

CHERI-Y86 is formalized using an interpreter approach to operational semantics and direct proofs verified using symbolic simulation based on Boyer-Moore clock functions [29]–[31]. A machine state object is defined in the ACL2 logic with field access and update functions. Semantic functions for each instruction in the ISA are implemented using the state object's access and update functions. A step function is defined to handle the fetch-decode-execute cycle, calling the appropriate instruction semantic functions during the execute stage. A run function is defined to take a number `n` of instructions to be executed and an initial CHERI-Y86 state object, and attempts to execute `n` calls to the step function.

A. The CHERI-Y86 State Object

We use a *single-threaded object* (stobj) to represent the CHERI-Y86 state in ACL2 [32]. The technical details of stobjs are beyond the scope of this paper, but the upshot is that stobjs enable extremely efficient field updates, and therefore fast execution speeds, by way of destructive memory assignments.

The fields of the CHERI-Y86 state `y86-64` are summarized in Table I. Those familiar with the Y86 architecture will find the CHERI-Y86 state machine similar, but some fields deserve extra comments. The first field is a static array of 16 general purpose registers. These registers are 128-bit in order to accommodate 128-bit capabilities. CHERI systems in general make a distinction between general-purpose registers and *capability registers*. However, the CHERI-x86-64 intends to extend existing general-purpose 64-bit x86 registers to 128-bit in order to store capabilities in the CHERI Concentrate format. We likewise accommodate 128-bit capabilities by extending general-purpose 64-bit Y86 registers to 128-bit. The Y86 model is intended to represent a 2^{64} byte physical memory and we maintain the byte-level semantics specified by the x86 specifications. The model state (MS) is a flag representing the state of the model as a simulator in ACL2 and not part of the original Y86 processor. It is used to signal problems with the model, such as when the CHERI-Y86 processor is halted. While the flag is `nil`, there is no problem with the CHERI-Y86 processor model; otherwise, the MS flag indicates the problem.

Each field of a model can be accessed by passing the CHERI-Y86 object corresponding to the model and invoking

the appropriate field. For example `(rip y86-64)` returns the program counter of `y86-64`. Fields with multiple parameters can be similarly accessed by passing the desired key or address. For example, to access the 0th register `rax`, use `(rgfi 0 y86-64)` or `(rgfi :rax y86-64)`. Updating the fields performed by prefixing `!` to the respective read commands. For example, to reset the program counter of `y86-64` to 0, invoke `(!rip 0 y86-64)`. Similarly, `(!rgfi :rcx #xB0BA7EA y86-64)` updates the 128-bit register `rcx` with 185313258.

B. Symbolic Simulation

Designed for symbolic or functional simulation, our simulator allows users to express machine state and instruction streams in symbolic ACL2 representations, enabling detailed, step-by-step observation of the model's behavior—including permission checks, capability manipulations, and error signaling. By working with interpretable, symbolic input and output, we lower the barrier for users to explore, reason about, and validate CHERI mechanisms within our x86-like model. While the current implementation does not natively load compiled binaries, extending the simulator to do so would primarily involve building pre- and post-processing layers that translate between binary formats and these symbolic representations. This design choice keeps our core model transparent and amenable to formal verification, while leaving open the possibility of direct binary support in future work.

C. Step & Run Functions

Simulation of machine program execution is performed by a sequence of calls to the CHERI-Y86 step functions. Each step function fetches an instruction from the memory of the CHERI-Y86 state object, decodes it, and then initiates the execute stage. Execution of the instruction is performed by semantic functions, which interpret the intended behaviour of the state under the instruction's effects and transforms the CHERI-Y86 processor from one state to the next. Modifications to the model state are permitted only if the appropriate capabilities are in place. The step function `y86-step` can be called on a state object `y86-64` with `(y86-step y86-64)`, which returns a new model state object after performing a fetch-decode-execute cycle. Note that this new state object may contain an error flag. Passing the number of steps desired to the `y86` function, e.g. `(y86 y86-64 n)` will attempt to run `y86-step n` times. If the run function encounters a state object with the MS flag raised, then `y86` immediately returns the state object. Typically, the MS flag indicates a model error or that a HALT instruction has been encountered. Otherwise, `(y86 y86-64 n)` will return the `y86-64` object in the state after `n` instructions have been executed.

D. CHERI-Y86 Instruction Semantic Functions

CHERI-Y86 instruction semantic functions specify the effect of machine code instructions on the CHERI-Y86 state. Our modelling of the Y86 instructions is comprehensive

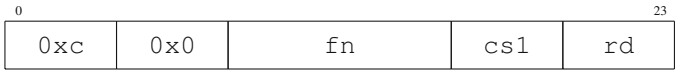

Fig. 3. CHERI-Y86 capability-inspection instruction format.

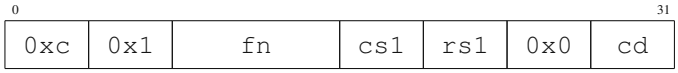

Fig. 4. CHERI-Y86 capability-modification instruction format.

and we implement the CHERI-Y86 equivalent of all non-deprecated instructions that are in the CHERI-x86-64 ISA v8 specifications. Because the latest CHERI-x86-64 specifications are merely a sketch, we take implementation inspiration from the CHERI-RISC-V where necessary.

We develop our own format for CHERI-Y86 capability-aware instructions while adhering to CHERI design goals. Figure 3 contains a visual representation of the CHERI-Y86 *capability-inspection* instruction format. A capability-inspection instruction reads the field of a capability stored in capability register `cs1` and writes the field to register `rd`. All CHERI-Y86 registers have been extended to support storing 128-bit capabilities, so `cs1` simply refers to a general-purpose register as well. We prefix instructions unique to CHERI with `0xc` to indicate they are capability-aware. The `fn` field in Figure 3 contains one of nine possible values since there are nine capability-inspection instructions described in the CHERI-x86-64 specification. Similarly, we've implemented all the capability-modification, pointer-arithmetic, and pointer-comparison instructions possible for a Y86-based architecture.

As an illustration, we briefly describe the implementation of GCPERM, a capability-inspection instructions which reads the permissions of a capability and stores them in a register. Algorithm 1 describes the GCPERM instruction with explanations inline. Part of the decode process is handled by the instruction semantic functions, namely the parts that handle reading data contained within the instruction such as source / destination registers, immediate values, memory addresses, etc. The assignments to `pc`, `rArB`, `rB`, and `rA` in Algorithm 1 are a partial consequence of the decode process. Identifying the particular instruction itself is handled by the step function, since it needs to dispatch execute responsibilities to the appropriate semantic function. The capability to be inspected `cs1` is stored in register `rA` in CHERI Concentrate format. The call to GETCAPABILITY is our pseudocode way of calling an interface function which decodes `cs1` in CHERI Concentrate format into an architectural capability. Assigning the decoded capability to `cs1` permits us to access the architectural capability fields with functions such as GCPERM, which reads the permissions field. Finally, updates are made to the CHERI-Y86 state object and the state object is returned.

Notice that state objects with the model state flag raised are returned when the instruction attempts to perform an unintended task. In Algorithm 1, this happens when the program counter is at an address in memory that would not

fit the size of the instruction, or when an attempt to read from register 15 is made, which is prohibited in Y86. For more complicated instructions, such as capability-modification instructions, setting the model state flag plays a much larger role. For example, CHERI protection features specify that capabilities should follow a kind of "monotonicity" property, which we will describe in Section V. The upshot is that should an instruction attempt to violate a CHERI architectural security property, then the semantic function will return a state object with a model state flag raised and an error message describing the property violated. Because these security properties are built into the *definition* of the semantic functions, *proving* that our model satisfies the CHERI protection features amounts to unwinding definitions. We also describe our proofs in Section V.

V. Verifying Basic CHERI-Y86 Properties

In this section, we describe some basic ACL2 features used to ensure the veracity of our CHERI-Y86 model and verify the CHERI protection features within. Rewrite rules are the method by which most ACL2 proofs are discharged. Rewrite rules enable the ACL2 system to replace subterms of a formula with other terms. Usually, rewrite rules are introduced into the ACL2 logic when a theorem is proven. Some theorems are automatically introduced into the logic whenever a new function is defined. For example, the *definition* of the function is a theorem itself. The rewrite rule associated with this theorem enables ACL2 to replace a call to the function with its definition, essentially "expanding" a function. A less trivial example is the theorem that states a newly defined function eventually terminates on all inputs. All functions in ACL2 must be proven to terminate before they are accepted into the logical universe. This is especially important for functions such as `y86` where avoiding the halting problem is desirable. In the case of `y86`, the function is *measured* by the number of steps it is given; that is, either `(y86 y86-64 n)` will terminate after `n` recursive calls or a model state error will be raised.

Some basic but important theorems of our model are *read-over-write* lemmas. These lemmas state that reading from a written field recovers the value written. For example, Listing 1 contains a theorem which states that writing a value `v` to register `i` in the `y86-64` state object, and then reading from register `i` again results in the same value `v`. Proving read-over-write lemmas are a good example of using rewrite rules. Expanding the definitions of `rgfi` and `!rgfi` exposes that underneath the stobj implementation are array-like data structures with their own read-over-write lemmas. The proof of `rgfi-!rgfi` in Listing 1 can be roughly thought of as the following sequence of rewrites

```
     (rgfi i (!rgfi i v y86-64))
⟶   (nth  i (!rgfi i v y86-64))
⟶   (nth  i (update-nth i v <rgfi-array>))
⟶   v
```

Similarly, there are theorems that describe fields to which no writes are performed. Suppose a write is made to memory

```
(defthm rgfi-!rgfi (equal (rgfi i (!rgfi i v
    y86-64)) v))
```

Listing 1. ACL2 read-over-write lemma for registers.

```
(defthm memi-read-through-different-address-!memi
  (implies (and (n64p i) (n64p j) (not (equal i j)))
           (equal (memi i (!memi j v y86-64))
                  (memi i y86-64))))
```

Listing 2. ACL2 theorem for memory invariance.

address `i`. It should follow that the memory at any address other than `i` is unchanged. Listing 2 contains the ACL2 theorem that states this property, which is discharged via rewriting as well. This property is perhaps even more important as it is a guarantee that contributes significantly to memory safety.

In addition to the read-over-write lemmas just described, there are also three more classes of theorems that together usually form a sufficient theory for proving more desirable theorems about a model:

1) write-over-write lemmas: two writes to different memory addresses can be performed in any order while resulting in the same final state object, and if the memory addresses were the same, then reading from the same memory address will result in the value of the most recent write;
2) write-over-read lemmas: writing a value that was just read from a memory address back into the same memory address will result in the same initial machine state;
3) state well formedness lemmas: writing a valid value to a field in a well formed machine state object results in another well formed state.

Basic protection features of CHERI capabilities are also proven using core ACL2 theorem proving features. For example, CHERI restricts capabilities that arise from other capabilities to an monotonicity property; that is, there is a partial order $\leq$ on capabilities, and if a capability-modification instruction is given a capability c_1 that results in a new capability c_2, then we must have $c_2 \leq c_1$. For example, consider CSetBounds, a CHERI-Y86 instruction that takes a capability c_1 and register r_2, and creates a new capability c_2 with bounds based on the address of c_1 and the value in r_2 (see Figure 4 for context). All other fields of c_2 are the same as c_1. In this case, we have $c_2 \leq c_1$ iff the bounds of c_2 are contained within the bounds of c_1, i.e. GetBounds$(c_2) \subseteq$ GetBounds(c_1). This is a reasonable CHERI protection feature; a capability should not be able to create a capability with greater memory access than its own. The way we prove this is simple:

1) initialize a well formed valid (no MS flag set) CHERI-Y86 state;
2) *symbolically* simulate one fetch-decode-execute cycle using CSetBounds;
3) prove GetBounds$(c_2) \subseteq$ GetBounds(c_1).

Performing step (3) amounts to applying rewrite rules the definition of CSetBounds. Recall the definition of GCPERM in Algorithm 1. When the instruction semantic

Algorithm 1 High-level description of CHERI-Y86 GCPERM instruction implementation.

```
procedure GCPERM(y86-64)
    pc ← RIP(y86-64)                                   ▷ Get PC from state
    if pc < 2^64 − 4 then                              ▷ Check if PC is too large
        return !MS(y86-64, "PC too large")             ▷ Return state w/ MS flag set & error description
    rArB ← READMEM(pc + 2, y86-64)                     ▷ Get the byte specifying src & dest registers
    rB ← rArB & 2^4                                    ▷ Get the dest register by masking upper 4 bits
    rA ← SHIFTRIGHT(rArB, 4) & 2^4 − 1                 ▷ Get the src register by shifting & masking
    if rA = 15 ∨ rB = 15 then                          ▷ Check if prohibited register specified
        return !MS(y86-64, "Prohibited register")      ▷ Return state with MS flag set
    cs1 ← GETCAPABILITY(y86-64, rA)                    ▷ Get & decode capability from src register
    cPerms ← GETPERM(cs1)                              ▷ Get permissions from capability
    y86-64 ← !RGFI(rB, cPerms, y86-64)                 ▷ Store permissions in dest register & update state
    y86-64 ← !RIP(pc + 3, y86-64)                      ▷ Update PC in state
    return y86-64                                      ▷ Return updated state
```

function attempts to perform a prohibited action, the function returns a state object with the MS flag raised. Similarly, the semantic function for CSETBOUNDS returns a state object with the MS flag raised should c_2 be assigned bounds outside of c_1's bounds. Therefore, if a valid CHERI-Y86 state object is returned after the fetch-decode-execute cycle is performed in step (2), then we must have GETBOUNDS$(c_2) \subseteq$ GETBOUNDS(c_1) and thus $c_2 \leq c_1$.

VI. VERIFYING CHERI CONCENTRATE

Our approach to proving the properties in Section III-A makes heavy use of the symbolic simulation framework GL with case-splitting [33], [34]. GL supports model checking with both binary decision diagrams (BDDs) as ACL2 symbolic objects and external SAT solvers. We use BDDs as the back-end engine for GL. While case splits were necessary for some of the larger proofs, we found that GL BDDs were sufficient for our verification needs. Moreover, GL's BDD proof procedure is verified in ACL2, alleviating any extra soundness concerns that may arise from calling external tools.

The ACL2 proofs of the encode / decode properties 5-6 are straightforward applications of GL's default symbolic simulation event `def-gl-thm`. To illustrate the use of GL, recall property 6 states

6) $b_0 = b_1$ and $t_0 = t_1$ when $\ell_0 < 2^{12}$.

Listing 3 contains the ACL2 theorem for property 6, i.e. the top and base bounds are exactly recoverable when $\ell_0 < 2^{12}$. The expression associated with the `:hyp` key

```
(def-gl-thm decode-encode-equal-small-seg
 :hyp   (and (valid-addr-p addr base len)
             (valid-b-l-p base len)
             (< len (expt 2 *TW*)))
 :concl (equal (decode-compression
                 (encode-compression len base)
                 addr)
               (bounds (+ len base) base))
 :g-bindings '((base ,(gl::g-int 0 3 65))
               (len  ,(gl::g-int 1 3 66))
               (addr ,(gl::g-int 2 3 65))))
```

Listing 3. ACL2 theorem for property 6 using GL.

is the GL method of indicating the hypotheses of the theorem. In this case, `(valid-addr-p addr base len)` and `(valid-b-l-p base len)` simply check that `addr`, `base`, and `len` are appropriately "typed." The hypothesis `(< len (expt 2 *TW*))` just checks that $\ell_0 < 2^{12}$. The key `:concl` denotes the conclusion of the theorem. For property 6. this simply states that decoding the encoded `base` and `len` with respect to an `addr` returns the well-formed bounds given by exactly the original `base` and `len`. The `:g-bindings` are shape specifications which assign the individual bits of `base`, `len`, and `addr` to BDD variables. The assignment of `base` to `gl::g-int 0 3 65` indicates the bits of `base` are assigned to BDD variables indexed $0,\ 3,\ 6,\ \ldots,\ 192 = 3 \times (65 - 1)$. Similarly, the bits of `len` are assigned to BDD variables indexed $1,\ 4,\ 7,\ \ldots,\ 193$ and the bits of `addr` assigned to BDD variables indexed $2,\ 5,\ 8,\ \ldots,\ 194$. One sign bit is required to satisfy the integer type but is redundant. The "mixed" order of the variables are for performance purposes. Under the hypotheses and shape specifications, the "domain" of the theorem is finite (i.e. `base`, `addr`, and `len` are 64-bit) and thus the theorem is amenable to verification via exhaustive symbolic execution of the functions (i.e. the `encode-compression`, `decode-compression`, `+`, and `bounds` functions) in the theorem under the BDD expressions. Moreover, the hypothesis restricts ℓ to $\ell < 2^{12}$, giving it relatively few possible values relative to its full 64-bit width (see Figure 6 for a visual representation), making the problem tractable for automated procedures. GL performs this symbolic simulation and readily proves the desired theorem.

In contrast to properties 5-6, properties 1-4 all required parameterized case splitting. Listing 4 is the ACL2 theorem of the encode / decode property 1, which states

1) $b_0 \geq b_1$ for any b_0, t_0, and address.

Note that the case where $\ell < 2^{12}$ is already handled by property 6, so we may restrict ourselves to when $\ell \geq 2^{12}$. Instead of attempting the proof while assuming the full 64-bit width of the three integers of interest, we split the domains of `base`,

```
(def-gl-param-thm decode-encode-b-bound-len>2^12
    :hyp   (and (valid-addr-p addr base len)
                (valid-b-l-p base len)
                (<= (expt 2 *tw*) len))
    :concl (<= (bounds->base (decode-compression (encode-compression len base) addr))
               base)
    :param-bindings
    `((((low 12) (high 16)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      (((low 16) (high 20)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
      ...
      (((low 64) (high 65)) ,(gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65)))))
    :param-hyp (and (<= (expt 2 low) len) (< len (expt 2 high)))
    :cov-bindings (gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
```

Listing 4. ACL2 theorem for property 1 using GL with parameterized case splitting.

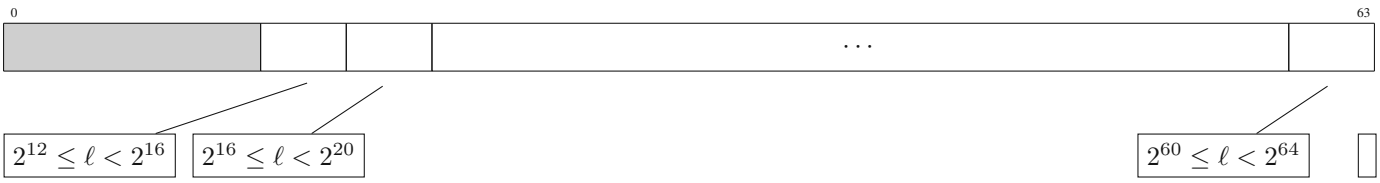

Fig. 5. GL case splits for theorem `decode-encode-b-bound-len>2^12` in Listing 4; cases must "cover" the domain $2^{12} \leq \ell < 2^{64}$.

Fig. 6. Case for theorem `decode-encode-equal-small-seg` in Listing 3: hypothesis restricts $\ell < 2^{12}$ under which GL proves the theorem.

`len`, and `addr` into 4-bit "intervals," effectively performing individual symbolic simulations for when the length ℓ satisfies

$$2^L \leq \ell < 2^H$$

for $L \in \text{LOW} \coloneqq \{4n : n \in \mathbb{N} \wedge 3 \leq n \leq 16\}$ and $H \in \text{HIGH} \coloneqq \{4n : n \in \mathbb{N} \wedge 4 \leq n \leq 16\} \cup \{65\}$, i.e. $2^{12} \leq \ell < 2^{16}$ and $2^{16} \leq \ell < 2^{20}$ and $2^{20} \leq \ell < 2^{24}$ and so on. In order to perform case splitting soundly, we must also show that our choice of splits for ℓ "covers" the intended domain of ℓ, i.e.

$$\begin{aligned}&\left\{\ell \in \mathbb{N} : 2^{12} \leq \ell < 2^{65}\right.\\ &\qquad \subseteq \left\{\ell \in \mathbb{N} : L \in \text{LOW} \wedge H \in \text{HIGH} \wedge 2^H \leq \ell < 2^H\right. \quad .\end{aligned}$$

Visually, this is represented by Figure 5. Note that the hypothesis stipulates that `len` is 64-bit, but we use an extra bit in the GL proof to make the bounds and coverage proof easier to state. The `:hyp` and `:concl` keys in Listing 4 indicate the hypothesis and the conclusion, respectively, similar to Listing 3. The key `:param-hyp` indicates *how* the case splitting should be performed; here, `len` is given various upper and lower bounds determined by different values of `low` and `high`. The values of `low` and `high` are given by the `:param-bindings`. GL "auto-bindings" are a convenient macro for defining the same shape specifications as in Listing 3 with the same "mixed" BDD variable ordering. In order for a proof by cases to be sound, we must also prove that the cases cover the domain of the theorem, which is why shape specifications are also provided to the `:cov-bindings` key.

Case splits for the CHERI Concentrate properties 2-4 are the same. Case splitting into smaller bit "intervals" would discharge the proofs faster at the expense of increasingly verbose user-provided hints. Case splitting being an effective approach suggests that the complexity of encoding and decoding does not scale with the length of the memory regions. Indeed, this accords with the design principles of CHERI Concentrate: "encoding efficiency, minimize delay of pointer arithmetic, and eliminate additional load-to-use delay" [28]. On the other hand, properties 5-6 required no case splitting largely because the encode / decode logic is greatly simplified when $\ell_0 < 2^{12}$, and alignment requirements reduce the number of variable bits.

VII. Conclusions and Future Work

We presented a formal model for the concurrent simulation and verification of CHERI architectural features. We extend Y86-64 with CHERI features because we intend to perform the same analysis on the x86-64 ISA augmented with CHERI features; however, the full specifications for CHERI-x86-64 have not yet been developed. Our model is built using the ACL2 programming language and verified with the ACL2 theorem prover. Function definitions double as functional specifications; many fundamental properties are verified via function expansion. For bit-twiddling proofs where hand-guiding the theorem prover was too cumbersome, we offload the work to GL, a fully ACL2 verified model checker with support for ACL2 BDDs. Our model comprises 9,856 lines of ACL2, including 2,446 lines of proof scripts. Key-property proofs typically completed in about two minutes each.

There are two major directions for future work. First, we intend to model CHERI-x86-64. CHERI-x86-64 is still in development; there is an opportunity to leverage existing ACL2 frameworks toward the new frontier of CHERI systems. Second, our model can be used to verify machine code programs involving CHERI instructions. In this paper, we verified straightline programs involving capability-aware instructions to analyse the protection features of CHERI capabilities. We leave the verification of longer capability-aware machine code programs for the future.

REFERENCES

[1] R. N. Watson, S. W. Moore, P. Sewell, and P. G. Neumann, "An introduction to CHERI," University of Cambridge Computer Laboratory, Technical Report 941, September 2019. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf

[2] R. Grisenthwaite. (2022) Morello research program hits major milestone with hardware now available for testing. [Online]. Available: https://www.arm.com/company/news/2022/01/morello-research-program-hits-major-milestone-with-hardware-now-available-for-testing

[3] S. Amar. (2022) An armful of CHERIs. [Online]. Available: https://msrc.microsoft.com/blog/2022/01/an_armful_of_cheris/

[4] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, F. A. Fuchs, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, "Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 9)," University of Cambridge Computer Laboratory, Technical Report 987, September 2023. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf

[5] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed. USA: Pearson, 2015.

[6] M. Kaufmann and J. Moore, "An industrial strength theorem prover for a logic based on Common Lisp," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, April 1997.

[7] J. B. Dennis and E. C. V. Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, March 1966.

[8] H. M. Levy, *Capability-Based Computer Systems*. Bedford, MA: Digital Press, 1984.

[9] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, "Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 8)," University of Cambridge Computer Laboratory, Technical Report 951, October 2020. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf

[10] J. Shapiro, J. Smith, and D. Farber, "EROS: A fast capability system," in *Proceedings of the Seventeenth Symposium on Operating System Principles*. ACM, 1999, pp. 170–185.

[11] J. Shapiro and J. Adams, "Coyotos microkernel specification, Version 0.6+," https://hydra-www.ietfng.org/capbib/cache/shapiro:coyotosspec.html, 2007.

[12] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: From general purpose to a proof of information flow enforcement," in *Symposium on Security and Privacy*. IEEE, 2013, pp. 415–429.

[13] W. Boebert, "On the inability of an unmodified capability machine to enforce the *-property," in *Proceedings of the 7th DoD/NBS Computer Security Conference*, September 1984, pp. 291–293.

[14] J. Shapiro and S. Weber, "Verifying the EROS confinement mechanism," in *Proceedings of the 2000 Symposium on Security and Privacy*. IEEE, 2000, pp. 166–176.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "SeL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. New York: Association for Computing Machinery, 2009, p. 207–220.

[16] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290384

[17] K. Nienhuis, A. Joannou, T. Bauereiss, A. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell, "Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1003–1020.

[18] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson, and P. Sewell, "Verified security for the morello capability-enhanced prototype arm architecture," in *Programming Languages and Systems*, I. Sergey, Ed. Cham: Springer International Publishing, 2022, pp. 174–203.

[19] S. Amar, D. Chisnall, T. Chen, N. W. Filardo, B. Laurie, K. Liu, R. Norton, S. W. Moore, Y. Tao, R. N. M. Watson, and H. Xia, "Cheriot: Complete memory safety for embedded devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 641–653. [Online]. Available: https://doi.org/10.1145/3613424.3614266

[20] A. L. Georges*, A. Guéneau*, T. Van Strydonck, A. Timany, A. Trieu*, D. Devriese, and L. Birkedal, "Cerise: Program verification on a capability machine in the presence of untrusted code," *J. ACM*, vol. 71, no. 1, Feb. 2024. [Online]. Available: https://doi.org/10.1145/3623510

[21] T. Van Strydonck, A. L. Georges, A. Guéneau, A. Trieu, A. Timany, F. Piessens, L. Birkedal, and D. Devriese, "Proving full-system security properties under multiple attacker models on capability machines," in *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, ser. 2022 IEEE 35th Computer Security Foundations Symposium (CSF), Haifa, Israel, Aug. 2022. [Online]. Available: https://inria.hal.science/hal-03826851

[22] J. Moore, T. Lynch, and M. Kaufmann, "A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm," *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 913–926, September 1998.

[23] W. A. Hunt, Jr., M. Kaufmann, J. S. Moore, and A. Slobadova, "Industrial hardware and software verification with ACL2," in *Verified Trustworthy Software Systems: Philosophical Transactions A, vol 374*, P. Gardner, P. O'Hearn, M. Gordon, G. Morrisett, and F. Schneider, Eds. Royal Society Publishing, 2017.

[24] S. Goel, "Formal verification of application and system programs based on a validated x86 ISA model," Ph.D. dissertation, University of Texas at Austin, 2016.

[25] A. Coglio and S. Goel, "Adding 32-bit mode to the ACL2 model of the x86 ISA," in *15th International Workshop on the ACL2 Theorem Prover and its Applications*, 2018.

[26] S. Goel, W. A. Hunt, Jr., and M. Kaufmann, "Engineering a formal, executable x86 ISA simulator for software verification," in *Provably Correct Systems (ProCos)*, 2017.

[27] S. Goel and W. A. Hunt, Jr., "Automated code proofs on a formal model of the x86," in *Automated Code Proofs on a Formal Model of the x86 (VSTTE)*, 2013.

[28] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. Norton, D. Chisnall, B. Davis, K. Gudka, N. Filard, A. Makettos, M. Roe, P. Neumann, R. Watson, and S. W. Moore, "CHERI concentrate: Practical compressed capabilities," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, October 2019.

[29] S. Ray and J. S. Moore, "Proof styles in operational semantics," in *Formal Methods in Computer-Aided Design*, A. J. Hu and A. K. Martin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 67–81.

[30] S. Ray, W. A. Hunt, J. Matthews, and J. S. Moore, "A mechanical analysis of program verification strategies," *Journal of Automated Reasoning*, vol. 40, pp. 245–269, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:532016

[31] "Mechanized operational semantics," https://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/index.html, accessed 2024-04-15.

[32] "Stobj," https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/ACL2____STOBJ, accessed 2024-04-15.

[33] S. Swords and J. Davis, “Bit-blasting ACL2 theorems,” in Proceedings 10th International Workshop *on the ACL2 Theorem Prover and its Applications,* Austin, Texas, USA, November 3-4, 2011, ser. Electronic Proceedings in Theoretical Computer Science, D. Hardin and J. Schmaltz, Eds., vol. 70. Open Publishing Association, 2011, pp. 84–102.

[34] S. Swords, “Term-level reasoning in support of bit-blasting,” in Proceedings 14th International Workshop on the *ACL2 Theorem Prover and its Applications,* Austin, Texas, USA, May 22-23, 2017, ser. Electronic Proceedings in Theoretical Computer Science, A. Slobodova and W. Hunt, Jr., Eds., vol. 249. Open Publishing Association, 2017, pp. 95–111.