Bounded Model Checking of Industrial Code

by

Siddharth Priya

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

© Siddharth Priya 2021

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some figures, tables, and text are restated from the Verifying Verified Code paper [45] and Bounded Model Checking for LLVM paper [44] (submitted) with contributions from Xiang Zhou, Yusen Su, Prof. Yakir Vizel, Dr. Yuyan Bao and Prof. Arie Gurfinkel. Some of the described modifications to SeaHorn were implemented by Prof. Arie Gurfinkel. Some of the experiments using `aws-c-common` were conducted by Prof. Yakir Vizel, Xiang Zhou and Yusen Su.

Abstract

Bounded Model Checking (BMC) is an effective and precise static analysis technique that reduces program verification to satisfiability (SAT) solving. However, with a few exceptions, BMC is not actively used in software industry, especially, when compared to dynamic analysis techniques such as fuzzing, or light-weight formal static analysis. This thesis describes our experience of applying BMC to industrial code using a novel BMC tool SeaBMC. We present three contributions.

First, a case study of (re)verifying the `aws-c-common` library from AWS using SeaBMC— a novel BMC engine for SeaHorn— and KLEE which is a well known symbolic checking tool. This study explores the methodology from the perspective of three research questions: (a) can proof artifacts be used across verification tools; (b) are there bugs in verified code; and (c) can specifications be improved. To study these questions, we port the verification tasks for `aws-c-common` library to SeaHorn and KLEE. We show the benefits of using compiler semantics and cross-checking specifications with different verification techniques, and call for standardizing proof library extensions to increase specification reuse.

Second, a description of SeaBMC. We start with a custom IR (called SEA-IR) that explicitly purifies all memory operations by explicating dependencies between them. We then run program transformations and allow for generating many different styles of verification conditions. To support memory safety checking, we extend our base approach with fat pointers and shadow bits of memory to keep track of metadata, such as the size of a pointed-to object. To evaluate SeaBMC, we use the `aws-c-common` library from AWS as a benchmark and compare with CBMC, SMACK, and KLEE. We show that SeaBMC can provide an order of magnitude improvement compared with state-of-the-art.

Third, a case study of extending SeaBMC to work with Rust— a young systems programming language. We ask three research questions: (a) can SeaBMC be used to verify Rust programs easily; (b) can the specification style of `aws-c-common` be applied successfully to Rust programs; and (c) can verification become more efficient when using higher level language information. We answer these questions by verifying aspects of the Rust standard library using SeaUrchin, an extension of SeaBMC for Rust.

Acknowledgements

I would like to thank Prof. Arie Gurfinkel for teaching me about software verification through countless one-on-one sessions. Moreover, his guidance throughout my MASc program has improved my critical thinking and presentation skills. For this, I am very grateful to him. I would like to thank Prof. Werner Dietl and Prof. Mahesh V. Tripunitara for readily agreeing to read my thesis and for their insightful questions during the seminar. Prof. Dietl painstakingly documented many errors in a draft version of this document. I appreciate his careful review of my work.

During my MASc, I learnt a lot working with my collaborators – Xiang Zhou, Yusen Su, Dr. Yuyan Bao and Prof. Yakir Vizel. Thank you for enriching my research experience. Because of the pandemic, I worked from home for most of my master's program but I did not feel isolated thanks to weekly chats with fellow student Hari Govind V.K. – who was always generous with his time. Before starting my master's, I grew as an engineer thanks to the mentorship provided over many years by Niket Agarwal – my gratitude to him.

Many years ago, my uncle, Sanjay Chandra gifted me a Lego technic kit which started my journey into how things work. I am grateful to him for giving me my first engineering project. I am grateful to Rishika and Mario for always being great listeners and asking insightful questions. Suyash helped me slow down and laugh when things got busy – thank you for the late night chitchats. I am indebted to my wife, Yashi, for being an inspiration and supporting me in my journey as a graduate student.

Dedication

This thesis is dedicated to my parents who have always encouraged me on my journey of curiosity. I also dedicate it to my children, Bhoomi and Avni. Hopefully, to them, my experience provides evidence that it's always worthwhile to pursue one's dreams.

Table of Contents

List of Figures ix

List of Tables xi

1 Introduction 1

2 Verifying Verified Code 3

2.1 Introduction 3

2.2 Unit Proofs with Code-as-Specification 6

2.3 Case Study 7

2.3.1 RQ1: Does CaS Empower Multiple Tools? 8

2.3.2 RQ2: Are there bugs in verified code? 11

2.3.3 RQ3: Can specifications be improved while maintaining the CaS philosophy? 13

2.4 Related work 16

2.5 Conclusion 17

3 SeaBMC: Bounded Model Checking using SeaHorn 19

3.1 Introduction 19

3.2 Verification Condition Generation 21

3.2.1 SEA-IR 21

3.2.2 Program Transformation 23

3.2.3 Verification Condition Generation 26
3.3 Verifying Memory Safety 29
3.3.1 Spatial memory safety 32
3.3.2 Temporal memory safety 33
3.4 Experiments 33
3.5 Related Work 35
3.6 Conclusion 35

4 SeaUrchin: Bounded Model Checking for Rust 37

4.1 Introduction 37
4.2 RQ1: Does LLVM bitcode as IR empower SeaBMC to work for Rust out-of-box? 38
4.3 RQ2: Does CaS philosophy apply effectively to Rust? 41
4.3.1 Panic freedom in Rust programs 41
4.4 RQ 3: Can verification become more efficient when using Rust language specific information? 44
4.5 Conclusion 48

5 Conclusion 50

References 52

List of Figures

2.1 The unit proof of `aws_array_list_get_at_ptr` from [11]. The commented out code is explained in Sec. 2.3.1. 7

2.2 Architecture of the case study (credit: Yusen Su). 8

2.3 Tool-specific implementations for `initialize_byte_buf`. 10

2.4 Simplified code for specification bugs. 11

2.5 Simplified code for differing CaS specifications. 13

2.6 Two styles of specifications for a read only buffer operation. 13

2.7 Linked list stubs for proofs. 15

2.8 SeaHorn unit proof for `aws_linked_list_front`. 15

3.1 Simplified grammar of SEA-IR, where `E`, `L` `R` and `M` are expressions, labels, scalar registers and memory registers, respectively. 23

3.2 An example C program. 23

3.3 Program from Fig. 3.2 in: (a) SEA-IR, (b) SA, (c) SASA, and (d) GSSA forms. 24

3.4 Program from Fig. 3.2 in PD and SMT-LIB forms. 26

3.5 Definition of *sym*. 27

3.6 Specifications for *size*, *align*, and *alloc*. 28

3.7 Translation of *read* and *write*. 28

3.8 SEA-IR syntax for memory safety. 29

3.9 Program from Fig. 3.2 with added **`isderef`** assertions in PD and SMT-LIB forms. 29

3.10 Memory-safety aware VCGen semantics. 30
3.11 Shadow memory semantics for memory safety. 30
3.12 Semantics for verifiying memory safety. 31
3.13 Memory state `M3`- when `P0` is stored at location `P1`. 31

4.1 Architecture of SeaUrchin. 39
4.2 String creation — panic and no-panic versions. 42
4.3 unit proof for mutable borrow. 44
4.4 Borrow for reading - panic and no-panic versions. 44
4.5 An equivalent program in Rust and C to write and read data from an array. 46
4.6 Comparision of verification time as array size increases for programs in Fig. 4.5. 47
4.7 The function `bong` calls an *unsafe* function. 48
4.8 The **`noalias`** function `bong` calls an *unsafe* function. 48

List of Tables

2.1 Verification results for CBMC, SeaHorn and KLEE (repro. from [45]). 7

3.1 Verification results for SeaBMC, CBMC, SMACK and KLEE. Timeout is 200s. **cnt**, **avg**, **std** and **time**, are the number of verification tasks, average run-time, standard deviation, and total run-time in seconds, per category, respectively (repro. from [44]). 34

Chapter 1

Introduction

Writing and shipping correct software remains a challenge in industry. The standard practice of testing can only improve the quality of software. It cannot provide any guarantees on correctness of software. Such guarantees are essential when software failures have economic and human costs. Software verification based on formal techniques can provide strong guarantees for software systems. Unfortunately, scaling these techniques to industrial grade software has been difficult. In particular we apply Bounded Model Checking (BMC), to industrial grade software. By scaling BMC, we mean three things: 1. large programs; 2. modelling of low-level instructions that occur in practice; and 3. verifying software systems instead of single programs — this may include more than one language. Our primary target is the `aws-c-common` library, written in C, open sourced by AWS. We discuss the results of verifying `aws-c-common` using SeaBMC, a new BMC engine developed for SeaHorn. We also present early work in applying BMC to Rust programs.

Bounded Model Checking (BMC) is an effective static analysis technique that reduces program analysis to propositional satisfiability (SAT) or Satisfiability Modulo Theories (SMT). It works directly on the source code. It is very precise, e.g., accounting for semantics of the programming language, memory models, and machine arithmetic. There is a vibrant ecosystem of tools from academia (e.g., SMACK [47], CPAChecker [8], ESBMC [22]), industrial research labs (e.g., Corral [36], F-SOFT [29]), and industry (e.g., CBMC [14], Crux [23], QPR [9]). There is an annual software verification competition, SV-COMP [7], with many participants. However, with a few exceptions, BMC is not actively used in software industry. Especially, when compared to dynamic analysis techniques such as fuzzing [48], or light-weight formal methods such as static analysis [6]. In Chapter 2, we report on our experience

of adapting the verification tasks of [11] to the new SeaBMC tool: a bit-precise Bounded Model Checking engine of SeaHorn. SeaBMC was developed hand in hand while verifying `aws-c-common`.

BMC can work on source code directly or on an Intermediate Representation (IR) of a compiler. SeaBMC works on IR of the LLVM [37] compiler. In Chapter 3, we look at the internals of SeaBMC. We look at two aspects broadly. First, we propose a new IR, called SEA-IR, that extends LLVM IR, with explicit dependency between memory operations. Different verification condition generation (VCGen) strategies operate on a program in SEA-IR form. Second, we explore different memory models which carry metadata about program state. Additional information is attached to pointers (so called *fat*) and to memory (so called *shadow*) to simplify tracking program metadata necessary for modelling many common safety properties. Our design, as implemented in SeaBMC, is evaluated on `aws-c-common` against state-of-the-art verification tools.

While C and C++ remain popular choices for writing system software, newer languages like Rust and GO are increasingly being considered for similar applications. Rust provides an interesting design choice where a program has to satisfy certain safety properties for compilation to be successful. Chapter 4 describes our on-going work to use SeaBMC to verify Rust programs. Rust programs contain more information than C because of a stronger type system. We want to explore if this information can help scale BMC to cover larger and more complex programs.

Chapter 2

Verifying Verified Code

2.1 Introduction

Transitioning research tools into practice requires case-studies, methodology, and best-practices to show how the tools are best applied. Until recently, there was no publicly available industrial case study on successful application of BMC for continuous verification[1] of C code. This has changed with [11] – a case study from the Automated Reasoning Group (ARG) at Amazon Web Services (AWS) on the use of CBMC for proving memory safety (and other properties) of several AWS C libraries. This case study proposes a verification methodology with two core principles: (a) verification tasks structured around units of functionality (i.e., around a single function, as in a unit test), and (b) the use of code to express specifications (i.e., pre-, post-conditions, and other contextual assumptions). We refer to these as *unit proofs*, and *Code as Specification (CaS)*, respectively. The methodology is efficient because small verification tasks help alleviate scalability issues inherent in BMC. More significantly, developers adopt, own, extend and even use specifications (as code) in other contexts, e.g., unit tests. Admirably, AWS has released all of the verification artifacts (code, specifications and verification libraries)[2]. Moreover, these are maintained and integrated into Continuous Integration (CI). This gave us a unique opportunity to study, validate, and refine the methodology of [11]. In this chapter, we report on our experience of adapting the verification tasks of [11] to two new verification tools: the SeaBMC Bounded Model Checking engine of SeaHorn,

[1] By *continuous verification*, we mean verification that is integrated with continuous integration (CI) and is checked during every commit.

[2] https://github.com/awslabs/aws-c-common/tree/main/verification/cbmc

and the symbolic execution tool KLEE. We present our experience as a case study that is organized around three Research Questions (RQ):

RQ1: Does CaS empower multiple tools for a common verification task? Code is the *lingua franca* among developers, compilers, and verification tools. Thus, CaS makes specifications understandable by multiple verification tools. To validate effectiveness of this hypothesis, we adapted the unit proofs from AWS to different tools, and report on the experience in Sec. 2.3.1. While giving a positive answer to RQ1, we highlight the importance of the semantics used to interpret CaS, and that effectiveness of each tool depends on specification styles.

RQ2: Are there bugs in verified code? Specifications written by humans may have errors. Do such errors hide bugs in verified implementations? What sanity checks are helpful to find bugs in implementations *and* specifications? The public availability of [11] is a unique opportunity to study this question. In contrast to [11], we found no new bugs in the library being verified (`aws-c-common`). However, we have found multiple errors in specifications! Reporting them to AWS triggered a massive review of existing unit proofs with many similar issues found and fixed. We report the bugs, and techniques that helped us discover them, in Sec. 2.3.2.

RQ3: Can specifications be improved while maintaining CaS philosophy? Some mistakes in specifications can be prevented by improvements to the specification language. We propose a series of improvements that significantly reduce specification burden. They are mostly in the form of built-in functions, thus, familiar to developers. In particular, we show how to make the verification of the `linked_list` data structure in `aws-c-common` significantly more efficient, while making the proofs unbounded (i.e., correct for linked list of any size).

In our case study, we used the BMC engine of SeaHorn [26] and symbolic execution tool KLEE [10]. We have chosen SeaHorn because it is conceptually similar to CBMC that was used in [11]. Thus, it was reasonable to assume that all verification tasks can be ported to it. We are also intimately familiar with SeaHorn. Thus, we did not only port verification tasks, but proposed improvements to SeaHorn to facilitate the process. We have chosen KLEE because it is a well-known representative of symbolic execution – an approach that is the closest alternative to Bounded Model Checking.

Overall, we have ported all of the 169 unit proofs of `aws-c-common` to SeaHorn, and 153 to KLEE. The case study represents a year of effort. The time was divided between porting verification tasks, improving SeaHorn to allow for a better comparison, and, many manual and semi-automated sanity checks to increase confidence in specifications. Additionally, we have experimented with using unit proofs as fuzz targets using LLVM fuzzing library `libFuzzer` [48] and adapted 146 of the unit proofs to `libFuzzer`.

We make all results of our work publicly available and reproducible at https://github.com/seahorn/verify-c-common. In addition to what is reported in this chapter, we have developed an extensive CMake build system that simplifies integration of additional tools. The case study is *live* in the sense that it is integrated in CI and is automatically re-run nightly. Thus, it is synchronized both with the tools we use and the AWS library we verify.

We hope that our study inspires researchers to adapt their tools to industrial code, and inspires industry to release verification efforts to study.

Caveats and non-goals. We focus on the issues of methodology and sharing verification tasks between different tools. The tools that we use have different strengths and weaknesses. While they all validate user-supplied assertions, they check for different built-in properties (e.g., numeric overflow, undefined behaviours, memory safety). The goal is not to compare the tools head-to-head, or to find the best tool for a given task. We have not attempted to account for the differences between the tools. Nor have we tried to completely cover all verification tasks by all tools. Our goal was to preserve the unit proofs of [11] as much as possible to allow for a better comparison. For that reason, while we do report on performance results for the different tools, we do not describe them in detail. An interested reader is encouraged to look at the detailed data we make available on GitHub. Furthermore, while we have applied fuzzing to the unit proofs, we do not focus on effectiveness and applicability of static vs dynamic verification but only on the issues of methodology.

To summarize, we make the following contributions: (a) we validate that CaS can be used to share specifications between multiple tools, especially tools that share the same techniques (i.e., BMC), or tools with related techniques (i.e., BMC and Symbolic Execution); (b) we describe in details bugs that are found in verified code (more specifically, in specifications), some are quite surprising; (c) we suggest a direction to improve CaS with additional built-in functions that simplify common specification; and (d) we make our system publicly available allowing other researches

to integrate their tools, use it as a benchmark, and to validate new verification approaches on industrial code.

The rest of the chapter is structured as follows. Sec. 2.2 recalls the methodology of unit proof and CaS. And Sec. 2.3 presents the architecture of the case study and answers the three research questions. We discuss related work in Sec. 2.4 and offer concluding remarks in Sec. 2.5.

2.2 Unit Proofs with Code-as-Specification

In [11], a methodology for program verification is proposed that allows developers to write specifications and proofs using the C programming language. The core of the methodology are *unit proof*[3] and *Code as Specification (CaS)*. A *unit proof* is similar to a *unit test* in that it is a piece of a code (usually a method) that invokes another piece of code (under test) and checks its correctness [43]. Fig. 2.1 shows an example of a unit proof for the method `aws_array_list_get_at_ptr`, from `aws-c-common` library. It has three parts: (1) the specification of `aws_array_list_get_at_ptr`, i.e., pre- (line 8) and post-conditions (lines 10–11); (2) a call to the function under verification (line 9); and (3) the specification of the program context that the method is called from (lines 2–7). Note that all specifications are written directly in C. We call this specification style – CaS. Assumptions (or pre-conditions) correspond to `__CPROVER_assume`, and assertions (or post-conditions) correspond to **`assert`**. Specifications are factored into functions. For example, `aws_array_list_is_valid` specifies a representation invariant of the array list. In this unit proof, the context is restricted to a list of bounded size but with unconstrained elements and an `index` with (intentionally) unspecified value of type **`size_t`**. Even without expanding the code further, its meaning is clear to any C developer familiar with the library.

The unit proof is verified with CBMC [14]. CBMC uses a custom SMT solver to check that there are no executions that satisfy the pre-conditions and violate at least one of the assertions (i.e., a counterexample). Together with the explicit assertions, CBMC checks built-in properties: memory safety and integer overflow.

According to [11], CaS and unit proofs are a practical and productive verification methodology. It has been used successfully to verify memory safety (and other properties) of multiple AWS projects, including the `aws-c-common` library that we use in our case study. The library provides cross platform configuration, data

[3] In [11], these are called *proof harnesses*.

```
1  void aws_array_list_get_at_ptr_harness() {
2    struct aws_array_list list;
3    /* memhavoc(&list, sizeof(struct aws_array_list))); */
4    __CPROVER_assume(aws_array_list_is_bounded(&list));
5    ensure_array_list_has_allocated_data_member(&list);
6    void **val = can_fail_malloc(sizeof(void *));
7    size_t index /* = nd_size_t() */;
8    __CPROVER_assume(aws_array_list_is_valid(&list) && val != NULL);
9    if (aws_array_list_get_at_ptr(&list, val, index) == AWS_OP_SUCCESS)
10     assert(list.data != NULL && index < list.length);
11   assert(aws_array_list_is_valid(&list)); }
```

Figure 2.1: The unit proof of `aws_array_list_get_at_ptr` from [11]. The commented out code is explained in Sec. 2.3.1.

		LOC			CBMC (s)		SeaHorn (s)		KLEE (s)		
category	num	avg	min	max	avg	std	avg	std	count	avg	std
arithmetic	6	33	11	40	3.8	0.8	0.6	0.1	6	0.9	0.3
array	4	97	78	112	5.6	0.0	1.7	0.7	4	32.3	6.0
array_list	23	126	77	181	35.8	60.8	2.5	3.3	23	55.4	49.3
byte_buf	29	97	50	188	17.6	47.3	1.0	0.8	27	75.3	124.1
byte_cursor	24	98	47	179	6.9	3.8	1.0	0.5	17	12.8	14.4
hash_callback	3	115	49	198	9.7	5.5	4.9	3.6	3	64.0	45.5
hash_iter	4	177	169	185	12.8	9.2	9.2	15.0	3	20.8	9.7
hash_table	19	172	36	328	23.5	33.3	5.3	7.5	15	104.6	333.4
linked_list	18	115	17	219	58.9	209.4	2.0	2.1	18	0.7	0.1
others	2	15	10	21	3.5	0.0	0.5	0.0	1	0.7	–
priority_queue	15	187	136	258	208.1	303.4	10.6	16.9	15	46.4	11.6
ring_buffer	6	155	56	227	20.0	19.5	29.5	34.2	6	48.1	26.4
string	15	87	11	209	6.3	1.3	2.9	1.8	15	139.7	159.7
Total	168	Loc	20,190		Time	6,475	Time	691	Time	8,577	

Table 2.1: Verification results for CBMC, SeaHorn and KLEE (repro. from [45]).

structures, and error handling support to a range of other AWS C libraries and SDKs. It is the foundation of many security related libraries, such as the AWS Encryption SDK for C [11]. It contains 13 data structures, 169 unit proofs that verify over 20K lines of code (LOC). Tab. 3.1 shows the LOC and running time for each data structure.

2.3 Case Study

The architecture of our case study is shown in Fig. 2.2. To compare with CBMC, we use two tools based on the LLVM framework [37]: SeaHorn and KLEE. SeaHorn [26] is a verification framework. We used the bit– and memory-precise BMC

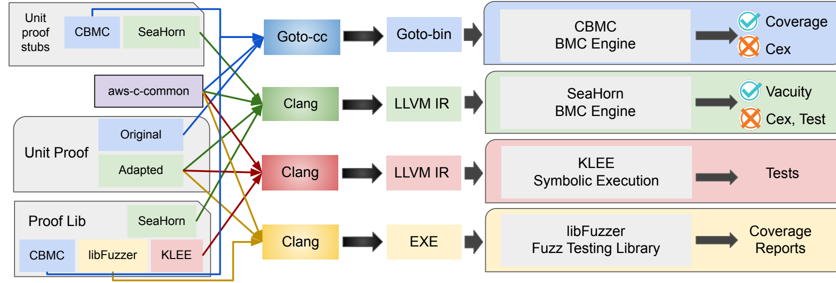


Figure 2.2: Architecture of the case study (credit: Yusen Su).

developed during the case study. Its techniques are closest to CBMC. KLEE [10] is a well-known symbolic execution tool. It is an alternative to BMC for bounded exhaustive verification. In addition, we have experimented with `libFuzzer` – a coverage-guided random testing framework. It does no symbolic reasoning, and, together with address sanitizer, is known to be effective at discovering memory errors. Fuzzing results are available online [4]. This section covers our experience primarily with SeaHorn. The details for KLEE and `libFuzzer` are given in our companion paper [45].

The rest of the section describes the research questions and our findings.

2.3.1 RQ1: Does CaS Empower Multiple Tools?

Hypothetically, the CaS methodology enables sharing the same formal specification among multiple, potentially distinct, tools and techniques. For example, semantic analyses of IDEs and compilers can catch simple semantics bugs and inconsistencies in specifications. Fuzzers can validate specifications through testing. Symbolic execution can supplement BMC by capitalizing on a different balance in performance versus precision. Static analysis tools can be used to compute inductive invariants. However, is the hypothesis true in practice?

To validate the hypothesis, we adapted the unit proofs from `aws-c-common` to two distinct verification techniques: BMC with SeaHorn and symbolic execution with KLEE. We have also attempted to use unit proofs as fuzz targets for `libFuzzer`. While our experience supports the hypothesis, we encountered two major challenges: *semantics* and *effectiveness of specifications*.

[4] https://seahorn.github.io/verify-c-common/fuzzing_coverage/index.html.

Semantics. Code without semantics is meaningless. Developers understand code without being versed in formal semantics, however, many technical details and "corner cases" are often debated. This is especially true for C – *"the semantics of C has been a vexed question for much of the last 40 years"* [40]. Clear semantics are crucial when code (and CaS) are used with multiple tools.

The unit proofs in [11] do not follow the C semantics. For example, consider the proof in Fig. 2.1. According to C, it has no meaning as both `list` (line 2) and `index` (line 7) are used uninitialized. CBMC treats uninitialized variables as non-deterministic. So it is well-defined for CBMC, but not for other tools.

What is a good choice of semantics for CaS? In [40], two semantics are described – the ISO C Standard and the *de facto* semantics of compilers. Developers understand (and use) the de facto semantics. For example, comparison of arbitrary pointers is undefined according to ISO C, but defined consistently in mainstream compilers (and used in `aws-c-common`!). Therefore, we argue that CaS must use the de facto semantics. Furthermore, unit proofs must be compilable and, therefore, executable, so developers can execute them not *just* in their heads (like with [11]). Note that de facto semantics is not complete with regards to C semantics, but is a commonly agreed upon subset. What de facto semantics does not cover is compiler dependent semantics.

In our experience, using CaS with the de facto semantics is not hard. For example, to adapt Fig. 2.1, we introduced `memhavoc` and `nd_size_t`, shown as comments, that fills a memory region at a given address with non-deterministic bytes, and returns a non-deterministic value of type **`size_t`**, respectively.

Effectiveness of specifications. We used three different tools on the same unit proofs. Each tool requires slightly different styles of specifications to be effective. We believe that these stylistic differences between specifications can be captured by traditional code refactoring techniques (i.e., functions, macros, etc.). However, this is not easy whenever the specifications have not been written with multiple tools (and with their strengths and weaknesses) in mind. A significant part of our work has been in refactoring unit proofs from [11] to be more modular.

We illustrate this with the pre-condition for the `byte_buf` data-structure. In [11], data structures are assumed to be initially non-deterministic, and various assumptions throughout the unit proof are used to restrict it (e.g., lines 2–5 in Fig. 2.1). This impedes specification re-use since different tools work well with different styles of pre-conditions. For example, symbolic execution and fuzzing require memory to

```
1       size_t len = nd_size_t();
2       size_t cap = nd_size_t();
3       assume(len <= cap);
4       assume(cap <= MAX_BUFFER);
5
6       buf->len = len;
7       buf->capacity = cap;
8       buf->buffer = can_fail_malloc(
9         cap * sizeof(*(buf->buffer)));
10      buf->allocator = sea_allocator();
11
12
13
14
15
```

(a) for SeaHorn

```
size_t cap = nd_size_t();
assume(cap <= MAX_BUFFER);
buf->buffer = can_fail_malloc(
  cap * sizeof(*(buf->buffer)));
if (buf->buffer) {
    size_t len = nd_size_t();
    assume(len <= cap);
    buf->len = len;
    buf->capacity = cap;
}
else {
    buf->len = 0;
    buf->capacity = 0;
}
buf->allocator = sea_allocator();
```

(b) for KLEE

```
size_t len = nd_size_t();
size_t cap = nd_size_t();
cap %= MAX_BUFFER;
len = (cap == 0) ? 0 : len % cap;

buf->len = len;
buf->capacity = cap;
buf->buffer = can_fail_malloc(
  cap * sizeof(*(buf->buffer)));
buf->allocator = sea_allocator();
```

(c) for `libFuzzer`

Figure 2.3: Tool-specific implementations for `initialize_byte_buf`.

be explicitly allocated, and all tools that use de-facto semantics require all memory be initialized before use.

For `byte_buf`, we factored out its pre-conditions into a function `init_byte_buf`.[5] Fig. 2.3, reproduced from our companion paper [45], shows its implementations for SeaHorn, KLEE, and `libFuzzer`. It takes `buf` structure as input, and initializes its fields to be consistent with the representation invariant of `byte_buf`.

SeaHorn initialization is closest to the original of [11]. Fields are initialized via calls to external functions (`nd_<type>`) that are assumed to return arbitrary values. Representation invariants (i.e., length is less or equal to capacity), as well as any upper bounds on buffer size are specified with *assumptions*. Note that `can_fail_malloc` internally initializes allocated memory via a call to `memhavoc`, ensuring that reading `buf->buffer` is well-defined.

Details for KLEE and `libFuzzer` are available in our companion paper [45]. Overall, our results indicate that CaS empowers multiple verification tools to share specifications among them. Common refactoring techniques make specifications sharing effective. Specifications are easiest to share among tools that use similar techniques.

Discussion. We conclude this section with a discussion of our experience in using de facto semantics. First, the code of `aws-c-common` is written with de facto semantics in mind. We found that in [11] it had to be extended with many conditional compilation flags to provide alternative implementations that are compatible with CBMC or that instruct CBMC to ignore some seemingly undefined behavior. However, we have not changed any lines of `aws-c-common`. We analyze the code

[5] Similarly, we introduced `init_array_list` to replace lines 2–5 in Fig. 2.1.

```
1  typedef
2  struct byte_buf {
3    char* buf;
4    int len, cap;
5  } BB;
6  bool BB_is_ok(BB *b)
7  { return (b->len == 0
8          || b->buf);  }
```

(a) bug 1

```
1  assume(0 <= b && b <= 10);
2  if (a < (b - 5) &&
3      a >= (b + 5))
4  {
5    assert(c > 0);
6  }
7
8
9
```

(b) bug 2

```
1  void ht_del_over(HASH_TB *t) {
2    /* remove entry */
3    /* t->entry_count--; */
4  }
5
6
7
8
```

(c) bug 3

Figure 2.4: Simplified code for specification bugs.

exactly how it is given to the compiler – improving coverage. Second, a compiler may generate different target code for different architectures. By using the compiler as front-end, we check that the code is correct as compiled on different platforms. This is another advantage of CaS. Third, compilers may provide additional safety checks. For example, `aws-c-common` uses GCC/Clang built-in functions for overflow-aware arithmetic. By using de facto semantics, all the tools used in the case study were able to deal with this in both CaS and code seamlessly. Fourth, `aws-c-common` uses inline assembly to deal with speculative execution-based vulnerabilities [33]. While inline assembly is not part of the ISO C standard, it is supported by compilers. Thus, it is not a problem for `libFuzzer`. We developed techniques to handle inline asm in SeaHorn. For KLEE, we had to ignore such unit proofs.

2.3.2 RQ2: Are there bugs in verified code?

Specifications may have errors as they are just programs: "Writing specifications can be as error-prone as writing programs" [42]. Although [11] suggests to use code coverage and code review to increase the confidence in specifications, we still found non-trivial bugs. We reproduce three bugs from our companion paper [45].

Bug 1. Fig. 2.4a shows the definition of `byte buf` that is a length delimited byte string. Its data representation should be either the buffer (`buf`) is `NULL` or its capacity (`cap`) is 0 (not the `len` as defined in `BB_is_ok`). We found this bug by a combination of sanity checks in SeaHorn and our model of the memory allocator (i.e., **`malloc`**). The bug did not manifest in [11] because other pre-conditions ensured that `buf` is always allocated. Our report of this bug to AWS triggered a massive code auditing effort in `aws-c-common` and related libraries where many related issues were found.[6]

[6] An example is https://github.com/awslabs/aws-c-common/pull/686/commits.

Bug 2. Fig. 2.4b shows a verification pattern where a property (line 5) is checked on the program path (from lines 1 to 5). As the condition at lines 2 and 3 can never be true, the property cannot be checked either. Our vacuity detection (discussed later) found the bugs occurring in this pattern. Note that the bug was missed by the code coverage detection used by CBMC, thus, may have been present for several years.

Bug 3. To make verification scalable, the verification of method A that calls another method B may use a *specification stub* that approximates the functionality of B. AWS adopts this methodology when verifying the iterator of a hash table. The iterator calls a function `ht_del` to remove an element in a hash table. During verification `ht_del` is approximated by a specification stub shown in Fig. 2.4c. However, the approximation does not decrement `entry_count`, i.e., line 3 should be added to the spec for correct behavior. In [11], the use of the buggy stub hides an error in the specification.

Discussion. Code coverage of a unit proof is, at best, a sanity check for CaS. It reports which source lines of the specification and code under verification are covered under execution. However, because source lines can remain uncovered for legitimate reasons e.g., dead code, interpreting a coverage report is not straightforward. There is no obvious *pass/fail* criterion. Thus, we found that code coverage may be insufficient to detect bugs in CaS reliably. In fact, bugs exist for multiple years even after code coverage failures. To help find bugs in CaS with SeaHorn, we adapted *vacuity detection* [35] to detect unreachable post-conditions. Vacuity detection checks that every assert statement is reachable. We encountered engineering challenges when developing vacuity detection. For example, we received spurious warnings due to code duplication. We silenced such warnings by only reporting a warning if all duplicate asserts reported a vacuity failure. In addition, due to CaS, an unreachable assertions may be removed by compiler's dead code elimination. This is not desirable for vacuity detection. To mitigate this issue, we report when dead code is eliminated. However, since many eliminations are unrelated to specs, there is noise in the report which makes it un-actionable. Interaction between dead code removal by the compiler and vacuity detection remains an open challenge for us.

We have found bugs in specifications, but we do not know what bugs remain. As shown in this section, the bugs were found with a combination of manual auditing and tools. However, these techniques are far from complete.

```
1  linked_list l;
2  Node *p = malloc(sizeof(Node));
3  l.head.next = p;
4  for (int idx=0; idx < MAX; idx++) {
5    Node *n = malloc(sizeof(Node));
6    p->next = n;
7    p = n; }
8  p->next = &l.tail;
9  l.tail.prev = p;
10 list_front(l);
11 Node *nnode = l.head.next;
12 for (int idx=0; idx < MAX; idx++) {
13   nnode = nnode->next;  }
14 assert(nnode == l.tail);
```

(a) Spec in the style of [11]

```
1  linked_list l;
2  Node *n = malloc(sizeof(Node));
3  n->next = nd_voidp();
4  l.head.next = n;
5  l.tail.prev = nd_voidp();
6  list_front(l);
7  assert(l.head.next == n);
```

(b) New specification

Figure 2.5: Simplified code for differing CaS specifications.

```
1    char buf[SZ];
2    init_buf(buf, SZ);
3    int idx = nd_int();
4    assume(0 <= idx && idx < SZ);
5    char saved = buf[idx];
6    read_only_op(buf);
7    assert(saved == buf[idx]);
```

(a) Spec in style of [11]

```
1    char buf[SZ];
2    init_buf(buf, SZ);
3    tracking_on();
4    read_only_op(buf);
5    assert(!is_mod(buf));
```

(b) Spec using a built-in `is_mod`

Figure 2.6: Two styles of specifications for a read only buffer operation.

2.3.3 RQ3: Can specifications be improved while maintaining the CaS philosophy?

There are many alternative ways to express a specification in CaS. In this section, we illustrate how to make proofs more efficient and make specs more readable. For example, a unit proof can fully instantiate a data structure (as in a unit test), or minimally constrain it (as in [11]). In this section, we illustrate this by describing our experience in making `linked_list` unit proofs unbounded (and more efficient). Furthermore, we believe that extending the specification language with additional verifier-supported built-in functions simplifies specs while making them easier to verify. We illustrate this with the built-ins developed for SeaHorn to specify absence of side-effects.

Linked List. A common pattern in *unit proofs* is to assume the representation invariant of a data structure, and to assert it after invocation of the function under verification along with other properties that must be maintained by the function. For example, a simplified version of its unit proof from [11] is shown in Fig. 2.5a. The pre-conditions are specified by (explicitly) creating a list in lines 4–7 using a loop. The post-condition is checked by completely traversing the list in lines 12–14.

This specification is simple since it closely follows the style of unit tests. However, it is inefficient for BMC: (a) unrolling the loops in the pre- and post-conditions blows up the symbolic search space; (b) it makes verification of the loop-free function `list_front` bounded, i.e., verification appears to depend on the size of the list in the pre-conditions.

Our alternative formulation is to construct a partially defined linked list stub as shown in Fig. 2.7a. This stub can be used to verify `list_front` since it is expected that only the first node after head is accessed. The resulting CaS is shown in Fig. 2.5b. The `next` field of `n` points to a potentially invalid address (returned by `nd_voidp`). Either `list_front` never touches `n->next` or has a memory fault. Finally, the assert on line 7 in Fig. 2.5b checks that `list_front` did not modify the head of the list either. If there is no memory fault, then `list_front` did not modify the linked list after the node `n`. Our specification is not inductive. It uses the insight that the given linked list API only ever accesses a single element. It, therefore, avoids loops in both the pre- and post-conditions and verifies `list_front` for linked lists of any size.

Unfortunately, our new spec in Fig. 2.5b is difficult to understand by non-experts because it relies on the interplay between `nd_voidp` and memory safety checking. To make the spec accessible, we hide the details behind a helper API. Fig. 2.8 shows the unit proof for `aws_linked_list_front` with this API. The function `sea_nd_init_aws_linked_list_from_head` constructs partial `aws_linked_list` instances with non-deterministic length (as shown in Fig. 2.7a). The function `aws_linked_list_save_to_` saves concrete linked list nodes from the partial `aws_linked_list`. Finally, the function `is_aws_list_unchanged_to_tail` is used in post-conditions to check that linked list nodes are not modified. The unit proof for `aws_linked_list_front` is not only more efficient than the original CBMC proof, but it is also a *stronger* specification. For example, if `aws_linked_list_front` removes or modifies a linked list node, our unit proof catches this as a violation, while the original proof only checks whether the returned value is valid and whether the linked list is well formed. The API we devised is generalized to work with all linked list operations in `aws-c-common`. For operations which access the node before the tail we construct a partially defined stub as shown in Fig. 2.7b while Fig. 2.7c is constructed for operations which access the list from both ends. We provide corresponding versions of the above API to save and check immutability of linked list nodes for each kind of stub.

Increasing CaS expressiveness. Verification tools should provide built-ins to aid in concise specifications. Moreover, such built-ins enable specifications that are

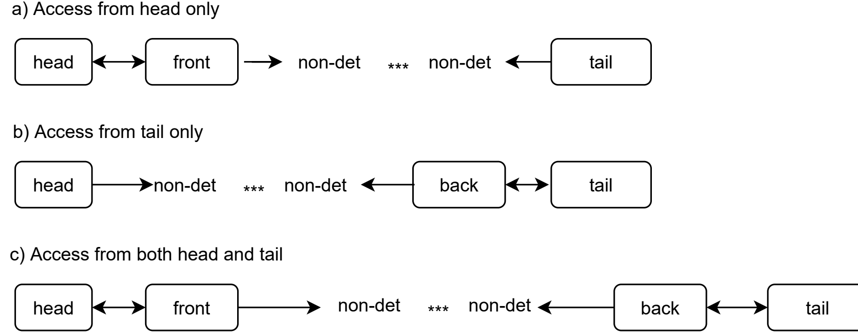


Figure 2.7: Linked list stubs for proofs.

```
1     void aws_linked_list_front_harness() {
2       /* data structure */
3       struct aws_linked_list list;
4       struct saved_aws_linked_list to_save = {0};
5       size_t size;
6
7       sea_nd_init_aws_linked_list_from_head(&list, &size);
8       struct aws_linked_list_node *start = &list.head;
9       aws_linked_list_save_to_tail(&list, size, start, &to_save);
10
11      // precondition in function does not accept empty linked list
12      assume(!aws_linked_list_empty(&list));
13
14      /* perform operation under verification */
15      struct aws_linked_list_node *front = aws_linked_list_front(&list);
16
17      /* assertions */
18      sassert(list.head.next == front);
19      sassert(aws_linked_list_node_prev_is_valid(front));
20      sassert(aws_linked_list_node_next_is_valid(front));
21      sassert(is_aws_list_unchanged_to_tail(&list, &to_save));
22
23      return 0;
24    }
```

Figure 2.8: SeaHorn unit proof for `aws_linked_list_front`.

not otherwise expressible in CaS. For example, Fig. 2.6b uses a SeaHorn built-in, `is_mod`, to specify that `read_only_op` does not change the buffer. This built-in returns true if memory pointed by its argument is modified since the last call to `tracking_on`. In contrast, the original specification for CBMC in Fig. 2.6a is tricky. It saves a byte from some position in the buffer (lines 3–5), and checks that it is not changed (line 7). This example illustrates that built-ins make specifications simpler and more direct. They ease specification writing for users and might be exploited efficiently by verification tools. As another example, SeaHorn provides a built-in `is_deref` to check that a memory access is within bounds, which is not (easily) expressible in C.

Discussion. CaS enables concise specifications and efficient proofs. As advanced verification techniques may not generalize, a standard extension is needed, such as verification-specific built-in functions. The semantics of these can be provided by a run-time library, validated by fuzzing and supported by multiple verification tools. Additional case studies are needed to identify a good set of built-ins. A standard extension can increase specification reuse and make verification more productive and effective.

2.4 Related work

To our knowledge, the recent study in [11] is the first significant, publicly available, example of an application of BMC on industrial code that is actively maintained with the code. Thus, our work is the first exploration of potential issues with software verified in this way. The closest verification case studies are `coreutils` with KLEE [10] and `busybox ls` with CBMC [32]. However, those focus on the scalability of a specific verification technology, while we focus on methodology, reuse, and what bugs might be hidden in the verification effort.

As we mentioned in the introduction, the Software Verification Competition (SV-COMP) [7] provides a large collection of benchmarks and an annual evaluation of many verification tools. However, it is focused on performance and soundness of the tools. The benchmarks are pre-processed to fit the competition format. At that stage, it is impossible to identify and evaluate the specifications, or to modify the benchmarks to increase efficiency of any particular tool. We hope that our case study can serve as an alternative benchmark to evaluate suitability of verification tools for industrial transition.

In addition to [11], there have been a number of other recent applications of BMC at AWS, including [13, 15, 16]. However, they are either not publicly available, too specialized, or not as extensive as the case study in [11].

Using code as specification has a long history in verification tools, one prominent example is Code Contracts [21]. One important difference is that in our case CaS is used to share specifications between completely different tools that only share the semantics of the underlying programming language, and the language itself is used to adapt specifications to the tools.

2.5 Conclusion

This case study would not have been possible without artifacts released by AWS in [11]. To our knowledge, it is the first publicly available application of BMC (to software) in industry. Related case studies on verification are those on `coreutils` with KLEE [10] and on `busybox ls` with CBMC [32]. SV-COMP is a large repository of benchmarks, but its goals are different from an actively maintained industrial project. The availability of both methodology and artifacts has given us a unique opportunity to study how verification is applied in industry and to improve verification methodology. We encourage industry to release more benchmarks to enable further studies by the research community.

In addition to answering the research questions, we are contributing a complete working system that might be of interest to other researchers. We have implemented a custom build system using CMake that simplifies integrating new tools. We provide Docker containers to reproduce all of the results. We created continuous integration (CI) on GitHub that nightly re-runs all the tools on the current version of `aws-c-common`. Since we use standard tools, the project integrates seamlessly into IDEs and refactoring tools. The CI runs are done in parallel by CTest. Running SeaHorn takes under 8 minutes!

While comparing different tools on performance is not our primary concern, in Tab. 3.1, we show the running time for all of the verification tools, collected on the same machine. For `libFuzzer`, we make the detailed coverage report available online. We stress that while the tools check the same explicit assertions, they check different built-in properties. Thus, running time comparison must be taken with a grain of salt.

Our main conclusion is in agreement with [11], and we strengthen the evidence for it. CaS is a practical and scalable approach for specifications that is easy to understand and empowers many tools. We argue that using de facto compiler semantics in CaS is key for enabling many verification tools, each with its own characteristic, to be used on the same verification problem. We find that specifications can be written in different ways and specification writer must account for both verification efficiency and developer readability. We suggest that a set of common built-ins be shared by different verification tools. Such built-ins improve the expressive power of CaS while retaining portability across verification tools. With built-ins defined in a specification library, software developers will be able to write unit proofs in a way no different than programming with libraries provided by some framework.

Today, formal verification is not the primary means of building confidence in

software quality. Our hope is that case studies like this one are useful to both software engineering researchers and practitioners who want to make formal methods an integral part of software development. To further this agenda, we plan to continue applying the CaS methodology to larger and more complex code bases (and languages) in the future.

Chapter 3

SeaBMC: Bounded Model Checking using SeaHorn

3.1 Introduction

Bounded Model Checking (BMC) is an effective technique for precise software static analysis. It works by encoding a bounded (i.e., loop- and recursion-free) program P with assertions into a verification condition VC in (propositional) logic, such that VC is satisfiable iff P has an execution that violates an assertion. The satisfiability of VC is decided by a SAT-solver (or, more commonly, by an SMT-solver). BMC can be extremely precise, including path-sensitivity, bit-precision, and precise memory model. Its main weakness is scalability – precise reasoning often requires careful selection of what details to include into analysis and what to abstract away.

It is possible to implement a BMC engine directly at the source level of a programming language. Best illustrated by CBMC [14] – perhaps the oldest and most mature BMC for C. This allows verifying the absence of undefined behaviour and other source-level properties, and improves error reporting since it can be done at the source level. However, this is very difficult to implement because modern programming languages are incredibly complex. Moreover, most industrial code uses de-facto, rather than the standard language semantics [40] and relies on non-standard features that are supported by common compilers. A second common alternative is to implement BMC on an intermediate representation (IR) of a compiler. IR of the LLVM [37] compiler, called *bitcode*, is a common choice. This simplifies implementation to focus only on capturing semantics of the IR, allows sharing infrastructure with the compiler, simplifies integration of verification into current build system, and simplifies

supporting multiple source languages (e.g., SMACK [47] supports 7 languages [24]). This is the approach we take in this chapter.

Over the years, there have been multiple BMC tools developed for LLVM, including SeaHorn (that we build on), SMACK, and LLBMC [41]. However, the issue still remains that existing tools are either not maintained, commercial (and not publicly available, e.g. LLBMC), or are not effective at bit- and memory-precise reasoning (SeaHorn and SMACK). Our goal is to address this deficiency, while re-examining and re-evaluating many of the design decisions.

While BMC is a mature technique, we believe that we identified a new interesting point in the design space. First, we propose a new IR, called SEA-IR, that extends LLVM IR with explicit dependency between memory operations. This, effectively, purifies memory operations, i.e., there is no global memory, and no side-effects. Second, we develop our verification condition generation (VCGen) as a series of program transformations. The program is progressively reduced to a pure data-flow form in which all instructions execute in parallel, and is only then, converted to SMT-LIB supported logic. This allows experimenting with different strategies of VCGen by controlling these transformations. Third, we explore two different forms of representing memory content: lambda-based that represents memory as nested ITE-expressions[1], and array-based that uses SMT theory of arrays. In particular, lambda-based representation allows precise and efficient modelling of wide memory operations such as `memcpy`. Fourth, we explore the space of memory models between the flat memory in which memory is a flat array, and an object memory model where memory is represented by a set of arrays. We settle on a representation in which each pointer operand of each memory instruction corresponds to an index of a unique array. Fifth, we attach additional information to pointers (so called *fat*) and to memory (so called *shadow*) to simplify tracking program metadata necessary for modeling many common safety properties.

We have evaluated SeaBMC on verification tasks of the `aws-c-common` C library developed by Amazon Web Services (AWS). The library is a collection of common data-structures for C (including buffers, arrays, lists, etc.). We chose it for several reasons. First of all, it has been recently verified using CBMC [11]. Thus, it includes many meaningful verification tasks. Second, it is a live industrial project, thus, it provides an example of how to integrate SeaBMC into a real project, and shows that SeaBMC supports all of the necessary language features. Third, it provides an opportunity to compare head-to-head against a mature tool (CBMC) on industrial code. We feel this is a more interesting comparison than, for exam-

[1]ITE stands for If-Then-Else.

ple, comparing on isolated verification benchmarks of SVCOMP [7]. We show that SeaBMC is sometimes an order of magnitude faster than CBMC. We also compare with two mature LLVM-based verification tools: SMACK and KLEE [10].

3.2 Verification Condition Generation

This section presents our main verification condition generation (VCGen) algorithm. We start with a new intermediate representation, that we call SEA-IR (Sec. 3.2.1). The representation extends LLVM bitcode with purified memory operations. We then describe a series of transformations that transform a program in SEA-IR to a pure data-flow (PD) form where no part of computation depends on control (Sec. 3.2.2). Finally, we show how PD programs can be converted to verification conditions in SMT-LIB (Sec. 3.2.3). In this section, we assume that the input program contains only one function, no loops or global variables. In practice, this is achieved by inlining all functions, unrolling loops to a fixed depth, and eliminating global variables. The loop unroll bound is often detected automatically, but can also be set by the user.

3.2.1 SEA-IR

SeaBMC transforms LLVM bitcode to an intermediate representation, called SEA-IR. In practice, SEA-IR is an extension of LLVM bitcode where dependency information between memory operations is made explicit. In LLVM IR, this information does not exist in the program. Fig. 3.1 shows the simplified syntax of SEA-IR. Here, we present a simplified version with many features removed, e.g., types, expressions, function calls, etc. However, we assume that the type of each register is known (but not shown). We use `R` to represent a scalar register, `P` for a pointer register and `M` for a memory register. A SEA-IR program is *well-formed* if it is in a Static Single Assignment (SSA) form, and satisfies other typical well-formedness conditions, such as all registers are defined before used, all expressions are well-typed, a program always ends with a **`halt`**, etc. We restrict the discussion to well-formed programs.

We use the term *object* to refer to an allocated sequence of bytes in memory. Interestingly, we do not use a single addressable memory that maps from addresses to values. Instead, a SEA-IR program uses a set of memory regions or *memories*, which collectively contains all objects in a program. Each memory, in-turn, contains a subset of objects used in the program. To maintain compatibility with de-facto semantics, addresses are assigned from a single address space and are, thus, globally

unique. To aid program analysis, all memories are pure: storing in memory creates a new memory i.e., *definition*; loading from a memory is a *use*. This def-use scheme [12] is known as MemorySSA in LLVM. Partitioning memory into multiple memories relieves the SMT-solver from some of the alias analysis reasoning.

To explain SEA-IR, we use a simple C program in Fig. 3.2. The program initializes variable `x` with a non-deterministic 8-bit integer obtained by the return value of function `nd_char()`. The value of x is further constrained by the **`assume`**, such that `x > 0 && x < 10`. Then, the program non-deterministically allocates a 1- or 2-byte memory region and assigns the address to the variable `p`. The first byte that p points to is assigned by the value of x. The second byte (if any) is assigned 0. For the moment, ignore that the second assignment might be undefined behaviour (we expand on this in Sec 3.3). Finally, the two **`assert`**s describe the post-condition.

Fig. 3.3a shows the SEA-IR program transformed from the C program. For presentation purposes, we do not strictly follow the syntax of SEA-IR. For example, we allow immediate values to appear in place of registers, and write expressions in infix form. The program is a single function **`main`**, which consists of four basic blocks labeled by `BB0`, `BB1`, `BB2` and `BB3`. A basic block consists of a label, zero or more `PHI`-statements, one or more statements, an optional branch statement or a **`halt`**.

A SEA-IR program has two types of registers: scalar registers and memory registers. Scalar registers store values of basic datatype – integers and pointers. Memory registers store memory regions, and map from addresses to values. Each memory register maps to a unique *memory* and we use memory register and memory interchangeably. For example, in Fig. 3.3a, `R0` is a scalar register which stores an integer and `P1` is a scalar register for a pointer. `M0` and `M1` are memory registers. Since each program is finite, the number of registers is finite as well.

An assignment statement defines the register by the value of a given expression. We assume that expressions include the usual set of operations, e.g., arithmetic, bitwise operations, cast operations and pointer arithmetic. For example, in `BB0` of Fig. 3.3a, `R2 = R0 < 10` defines the value of register `R2` by the value of the expression `R0 < 10`, where $<$ is an unsigned 8-bit less-than operator.

A **`phi`** selects a value from a list of values when a control flow merges. For example, `M3 =` **`phi`**`[M1,BB1],[M2,BB2]` in `BB3` of Fig. 3.3a assigns `M1` (`M2`) to `M3` if the previously executing was `BB1` (`BB2`).

SEA-IR provides **`alloca`** and **`malloc`** instructions to allocate memory on the stack and heap, respectively. A given number of bytes are allocated in memory on RHS of the statement, defining a new memory on the LHS. While allocation itself

$$
\begin{array}{rcl}
\mathrm{P} & ::= & \texttt{fun main()}\{\mathrm{BB}^{+}\} \\
\mathrm{BB} & ::= & \mathrm{L} : \mathrm{PHI}^{*}\ \mathrm{S}^{+}\ (\mathrm{BR} \mid \texttt{halt}) \\
\mathrm{BR} & ::= & \texttt{br}\ \mathrm{E},\ \mathrm{L},\ \mathrm{L} \mid \texttt{br}\ \mathrm{L} \\
\mathrm{PHI} & ::= & \mathrm{R} = \texttt{phi [}\mathrm{R},\ \mathrm{L}\texttt{](, [}\mathrm{R},\ \mathrm{L}\texttt{])}^{*} \mid \mathrm{M} = \texttt{phi [}\mathrm{M},\ \mathrm{L}\texttt{](, [}\mathrm{M},\ \mathrm{L}\texttt{])}^{*} \\
\mathrm{S} & ::= & \mathrm{RDEF} \mid \mathrm{MDEF} \mid \mathrm{VS} \\
\mathrm{RDEF} & ::= & \mathrm{R} = \mathrm{E} \mid \mathrm{R},\ \mathrm{M} = \texttt{alloca}\ \mathrm{R},\ \mathrm{M} \mid \mathrm{R},\ \mathrm{M} = \texttt{malloc}\ \mathrm{R},\ \mathrm{M} \mid \\
& & \mathrm{R} = \texttt{load}\ \mathrm{R},\ \mathrm{M} \mid \mathrm{M} = \texttt{free}\ \mathrm{R},\ \mathrm{M} \\
\mathrm{MDEF} & ::= & \mathrm{M} = \texttt{store}\ \mathrm{R},\ \mathrm{R},\ \mathrm{M} \\
\mathrm{VS} & ::= & \texttt{assert}\ \mathrm{R} \mid \texttt{assume}\ \mathrm{R}
\end{array}
$$

Figure 3.1: Simplified grammar of SEA-IR, where `E`, `L` `R` and `M` are expressions, labels, scalar registers and memory registers, respectively.

```
1  int main() {
2    uint8_t x = nd_char();
3    assume(x > 0 && x < 10);
4    uint8_t *p = nd_bool() ? malloc(2*sizeof(uint8_t))
5                           : malloc(sizeof(uint8_t));
6    *p = x;
7    *(p + 1) = 0; // potential UB
8    sassert(0 < *p && *p < 10);
9    sassert(*(p + 1) == 0); // potential UB
10   return 0;
11 }
```

Figure 3.2: An example C program.

does not define memory, the reason for this syntax is explained in Sec. 3.3. Consider `P1, M1 = malloc 2, M0` in `BB1` of Fig. 3.3a. It allocates 2 bytes (in heap address space) in memory `M0`, defines memory `M1` and a fresh pointer in `P1`.

A **`store`**, e.g., `M5 = store 0, R5, M4` in `BB3`, defines memory `M5` by writing the value 0 to the address pointed-to by the pointer register `R5` in memory `M4`. Note that the instruction is pure; i.e., all effects of the instructions are on the output registers only. The result of the modification is stored in `M5`, while `M4` is unchanged. Similarly, a **`load`** reads the value pointed-to by a pointer register in memory register `M`, and assigns the value to a new register. **`assert`** and **`assume`** represent the usual verification statements for assertions and assumptions, respectively.

3.2.2 Program Transformation

Before generating verification conditions, a series of program transformations are applied to a SEA-IR program. This section explains each program transformation.

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0
  assume R1
  R2 = R0 < 10
  assume R2
  R3 = nd_bool()
  br R3, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = phi [M1,BB1],[M2,BB2]
  R4 = phi [P1,BB1],[P2,BB2]
  M4 = store R0, R4, M3
  R5 = R4 + 1
  M5 = store 0, R5, M4
  R6 = R0 > 0 && R0 < 10
  assert R6
  assert 0 == 0
  halt
}
```

(a) SEA-IR

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0
  assume R1
  R2 = R0 < 10
  assume R2
  R3 = nd_bool()
  br R3, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = phi [M1,BB1],[M2,BB2]
  R4 = phi [P1,BB1],[P2,BB2]
  M4 = store R0, R4, M3
  R5 = R4 + 1
  M5 = store 0, R5, M4
  R6 = R0 > 0 && R0 < 10
  br R6, BB4, ERR
BB4:
  assume 0 != 0
  br ERR
ERR:
  assert 0
  halt
}
```

(b) Single Assert (SA)

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  br R2, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = phi [M1,BB1],[M2,BB2]
  R3 = phi [P1,BB1],[P2,BB2]
  M4 = store R0, R3, M3
  R4 = R3 + 1
  M5 = store 0, R4, M4
  R5 = R0 > 0 && R0 < 10
  br R5, BB4, ERR
BB4:
  R6 = false
  br ERR
ERR:
  A = phi [R6,BB4],[R1,BB3]
  assume A
  assert 0
  halt
}
```

(c) Single Assume (SASA)

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  br R2, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = select R2, M1, M2
  R3 = select R2, P1, P2
  M4 = store R0, R3, M3
  R4 = R3 + 1
  M5 = store 0, R4, M4
  R5 = R0 > 0 && R0 < 10
  br R5, BB4, ERR
BB4:
  R6 = false
  br ERR
ERR:
  A = select R5, R6, R1
  assume A
  assert 0
  halt
}
```

(d) Gated SSA (GSSA)

Figure 3.3: Program from Fig. 3.2 in: (a) SEA-IR, (b) SA, (c) SASA, and (d) GSSA forms.

Single Assert Form A program is in a Single Assert (SA) form if it only contains one **assert**, which appears as the last statement in the last block of a program before **halt**. Fig. 3.3b shows the code in a SA form transformed from the one in Fig. 3.3a, where an `ERR` label is added to the original code, and denotes an error state. In `BB3`, **assert** `R6` is transformed into **br** `R6, BB4, ERR`, meaning that if `R6` is not true, then the program's execution trace is diverted to `ERR`. Similarly, **assert** `0 = 0` in `BB3` is transformed into **assume** `0 != 0` and **br** `ERR`.

Single Assume Single Assert Form A program is in a Single Assume Single Assert (SASA) form if it is in SA form, and contains a single **assume** immediately followed by a single **assert**. For example, the two definition of registers `R1` and `R2` in `BB0` of Fig. 3.3b are combined into one definitions of `R1` in Fig. 3.3c, where the two boolean expressions are combined by a conjunction. A **phi**-statement, `A =` **phi** `[R6,BB4],[R1,BB3]`, is added to `ERR`. Thus, a register `A` tracks the value of the conjunction along an execution, and using the single **assume** to check the value in the register is true.

Gated Single Static Assignment Form A program in SASA form is further transformed into a Gated Single Static Assignment (GSSA) form, where **phi**-functions are replaced by **select** expressions[2]. For example, **phi** `[M1,BB1], [M2,BB2]` in `ERR` of Fig. 3.3c is transformed into **select** `R2, M1, M2` in Fig. 3.3d, where `R2` is the condition that the program trace is diverted to `BB1` or `BB2`.

Pure Dataflow Form A (loop-free) program is in a Pure Dataflow (PD) form if it is in GSSA form and contains a single basic block. As shown in Fig. 3.4a, all the labels and **br** are removed from Fig. 3.3d, and the five basic blocks are merged into one single basic block.

Reduced Pure Dataflow Form A program is in a reduced PD form if every definition appears on a def-use chain of either **assume** or **assert**. Each such definition is said to be in the cone of influence (COI). In Fig. 3.4a, the highlighted code is not in the cone of influence and is not considered.

A program in reduced pure data flow form has no control dependencies. It is essentially a sequence of equations with two side-conditions determined by **assume**

[2]In LLVM, **select** is the usual ternary ITE such as `c ? a : b` in C.

```
fun main() {
entry:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  P1, M1 = malloc 2, M0
  P2, M2 = malloc 1, M0
  M3 = select R2, M1, M2
  R3 = select R2, P1, P2
  M4 = store R0, R3, M3
  R4 = R3 + 1
  M5 = store 0, R4, M4
  R5 = R0 > 0 && R0 < 10
  R6 = false
  A = select R5, R6, R1
  assume A
  assert 0
  halt
}
```

(a) Pure-Dataflow (PD)

$r_1 = (r_0 > 0 \land r_0 < 10) \land$

$p_1 = addr_0 \land m_1 = m_0 \land$
$p_2 = addr_0 + 4 \land m_2 = m_0 \land$

$r_3 = ite(r_2, p_1, p_2) \land$

$r_4 = r_3 + 1 \land$

$r_5 = (r_0 > 0 \land r_0 < 10) \land$
$r_6 = false \land$
$a = ite(r_{=5}, r_6, r_1) \land$
$a \land$
$\neg false$

(b) SMT-LIB

Figure 3.4: Program from Fig. 3.2 in PD and SMT-LIB forms.

and **assert**. All definitions are used, directly, or indirectly, by either **assume** or **assert** (or both). At this point, generating VC is a matter of mapping each definition into an equation in a logic.

3.2.3 Verification Condition Generation

In this section, we describe the translation function *sym* that encodes a program into a verification condition. Throughout the section, we illustrate *sym* using the program in Fig. 3.4a and the corresponding VC in Fig. 3.4b.

The input to *sym* is a SEA-IR program in a reduced PD form, and the output is a SMT-LIB 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, $bv(64) \to bv(64)$.[3] In addition, we use the following helper sorts: $scalr : bv(64)$, $ptrs : scalr$, and $mems : bv(64) \to bv(64)$, where *scalr* is sorts of scalars, *ptrs* of pointers, and *mems* of memories.

sym is defined recursively, bottom up, on the abstract syntax tree of SEA-IR. First, each register, R, is mapped to a symbolic constant $sym(\text{R})$ of an appropriate sort. To simplify the presentation, we use a lower-case math font for constants

[3]In practice, SeaBMC supports multiple bit-widths for scalars, and different ranges for values for maps.

$$sym(\texttt{R = E}) \triangleq r = e \quad sym(\textbf{assume}\ \texttt{R}) \triangleq r \quad sym(\textbf{assert}\ \texttt{R}) \triangleq \neg r$$

$$sym(\texttt{M1 = }\textbf{store}\ \texttt{R1,R2,M0}) \triangleq m_1 = write(m_0, r_1, r_2)$$

$$sym(\texttt{R1 = }\textbf{load}\ \texttt{R0,M}) \triangleq r_1 = read(m, r_0)$$

$$sym(\texttt{R1,M1 = }\textbf{alloca}\ \texttt{R0,M0}) \triangleq r_1 = alloc(\textbf{alloca}\ \texttt{R0,M0}) \ \wedge\ m_1 = m_0$$

$$sym(\texttt{R1,M1 = }\textbf{malloc}\ \texttt{R0,M0}) \triangleq r_1 = alloc(\textbf{malloc}\ \texttt{R0,M0}) \ \wedge\ m_1 = m_0$$

Figure 3.5: Definition of *sym*.

corresponding to the register. For example, in Fig. 3.4a, sym(`R0`) is r_0 of *scalr* sort, sym(`P2`) is p_2 of *ptrs* sort, and sym(`M0`) is m_0 of *mems* sort, respectively.

Second, each expression `E` in SEA-IR is mapped into a corresponding SMT-LIB expression sym(`E`). We omit the details of this step since they are fairly standard. For example, a **`select`** is translated into an *ite*, scalar addition, such as `R9 + 1` is translated into bit-vector addition `bvadd`, etc. Pointer manipulating expressions, such as pointer arithmetic (**`gep`**) and pointer-to-integer cast (**`ptoi`**) are described in Sec. 3.3.

Finally, *sym* translates each statement into an equality. For example, `R = E` is translated into $r = e$, where e is sym(`E`). For example, in Fig. 3.4a, `A =` **`select`** `R5,R6,R1` is translated into $a = ite(r_5, r_6, r_1)$ in Fig. 3.4b.

Translating **`alloca`** and **`malloc`** requires a memory allocator. We parameterize *sym* by an allocation function $alloc : A \to ptrs$ that maps allocation expressions in A to values of pointer sort. For example, in Fig. 3.5, `R1, M1 =` **`alloca`** `R0, M0` is translated into $r_1 = alloc($**`alloca`** `R0 M0`$) \ \wedge\ m_1 = m_0$, and is reduced to $p_1 = addr_0 \wedge m_1 = m_0$, where $addr_0$ is the return value of *alloc*.

For *sym*, *alloc* must satisfy the basic specifications of a memory allocator. The spec is formalized in Fig. 3.6, where *size* and *align* return the size and alignment of each allocation expression in A. Intuitively, each allocated segment must have a statically known bound on size, all pointers returned by an allocation are aligned, and all allocations are mutually disjoint. For example, in Fig. 3.4a, the memory allocations in `P2, M1 =` **`malloc`** `2, M0` and `P1, M2 =` **`malloc`** `1, M0` are guaranteed to be disjoint since Fig. 3.4b adds a constraint that $p_1 = addr_0 \wedge p_2 = addr_0 + 4$. In practice, we also enforce that stack allocations (**`alloca`**) return high addresses, and heap allocations (**`malloc`**) return low addresses. Other constraints, such as separating kernel- and user-space addresses can be easily added.

The semantics for memory operations depends on the representation of memories (see Sec. 3.3). We use two functions, *read* and *write*, to encapsulate the actual translation when defining the meaning of **`load`** and **`store`**, respectively. The function

$$\forall a \in A \cdot \mathit{size}(a) \text{ is known} \quad \forall a \in A \cdot (\mathit{alloc}(a) \bmod \mathit{align}(a)) = 0$$
$$\forall a_1 \neq a_2 \in A \cdot (\mathit{alloc}(a_1) + \mathit{size}(a_1) \leq \mathit{alloc}(a_2)) \vee (\mathit{alloc}(a_2) + \mathit{size}(a_2) \leq \mathit{alloc}(a_1))$$

Figure 3.6: Specifications for *size*, *align*, and *alloc*.

	`Array`	λ
$\mathit{read}(m, r_0)$	**`select`** $m\ r_0$	$m(r_0)$
$\mathit{write}(m_0, r_1, r_2)$	**`store`** $m_0\ r_1\ r_2$	$\lambda x.\mathit{ite}(x = r_1, r_2, m_0(x))$

Figure 3.7: Translation of *read* and *write*.

$\mathit{read}(m, r)$ represents the value of the memory register m at index r. The function $\mathit{write}(m, r_1, r_2)$ represents a new memory obtained by writing the value r_1 at index r_2 in m. In Fig. 3.5, **`load`** `R0, M` and **`store`** `R1, R2, M0` are translated into $\mathit{read}(m, r_0)$, and $\mathit{write}(m_0, r_1, r_2)$, respectively.

SeaBMC has two memory representations:

Arrays Memories are modeled by an SMT-LIB theory of extensional arrays ArraysEx[4]. A memory register `M` is mapped to a symbolic constant m, where m is of sort *mems*. As shown in Fig. 3.7, a *write* is translated into an ArrayEx store, and a *read* is translated into an ArrayEx select.

Lambdas Memories are modelled by λ-functions of the form $\lambda x.e$, where e is an expression with free occurrences of x. A memory register `M` is translated into an uninterpreted function m of sort *mems*. As shown in Fig. 3.7 $\mathit{read}(m, r_0)$ is translated into a function application $m(r_0)$, and $\mathit{write}(m_0, r_1, r_2)$ is translated into a new λ-function, $\lambda x.\mathit{ite}(x = r_1, r_2, m_0)$. In the final VC, function applications are β-reduced to substitute formal arguments with actual parameters. Thus, the VC contains only *ite*-expressions, and does not require underlying SMT-solver to support ArrayEx.

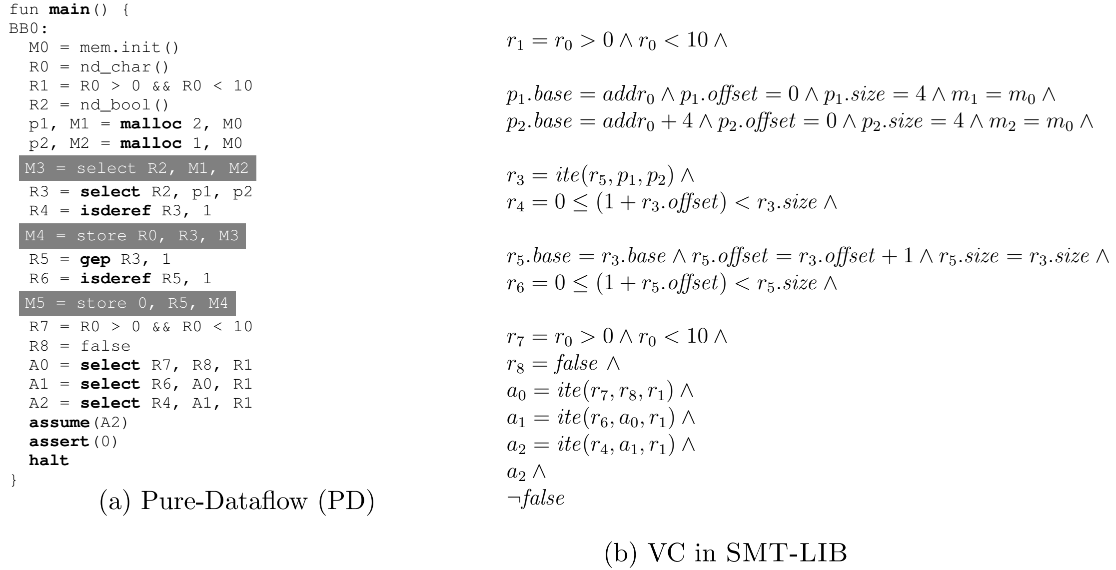
Overall, for a program `P` in a reduced PD form with a sequence of statements $\mathtt{S}_0 \cdots \mathtt{S}_k$, followed by **`assume`** `R0` and **`assert`** `R1`, $\mathit{sym}(\mathtt{P})$ is defined as follows:

$$\mathit{sym}(\mathtt{P}) \triangleq \left(\bigwedge_{0 \leq i \leq k} \mathit{sym}(\mathtt{S}_i) \right) \wedge \mathit{sym}(\mathtt{R0}) \wedge \neg \mathit{sym}(\mathtt{R1}).$$

[4] http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml.

$\text{RDEF} \quad ::= \quad \text{R} = \textbf{isderef}\ \text{R},\ \text{R} \mid \text{R} = \textbf{isalloc}\ \text{R}, \text{M} \mid \text{R} = \textbf{ismod}\ \text{R},\ \text{M}$

Figure 3.8: SEA-IR syntax for memory safety.



(a) Pure-Dataflow (PD)

(b) VC in SMT-LIB

Figure 3.9: Program from Fig. 3.2 with added **isderef** assertions in PD and SMT-LIB forms.

For example, the VC for a program in Fig. 3.4a is shown in Fig. 3.4b. Definitions in Fig. 3.4a are translated into a conjunction of equalities, and **assert** 0 is translated into $\neg false$. The VC is *unsatisfiable* since A evaluates to *false*.

Theorem 1 *$sym(\mathtt{P})$ is satisfiable iff* P *has an execution that satisfies the assumption and violates the assertion.*

3.3 Verifying Memory Safety

Memory safety is difficult to specify directly in many programming languages. In C, for example, there is no mechanism for checking if a pointer dereference is within allocated bounds. To make such specifications possible, we use fat pointers [30]

$$sym(\texttt{R1,M1 = }\textbf{malloc}\texttt{ R0,M0))} \triangleq r_1 = alloc(\textbf{malloc}\texttt{ R0, M0}) \wedge m_1 = alloc_{sh}(m_0, r_1)$$
$$sym(\texttt{M1 = }\textbf{free}\texttt{ R0,M0}) \triangleq m_1 = free_{sh}(m_0, r_0)$$
$$sym(\texttt{MR = }\textbf{store}\texttt{ R1,R2,M1))} \triangleq \langle m_{r1}, \ldots, m_{rj}\rangle = \langle write(m_0.1, addr(r_1), r_2.1), \ldots, write(m_0.j, addr(r_1), r_2.j)\rangle \wedge \langle m1_{j+1}, \ldots, m1_k\rangle = store_{sh}(\langle m_{0j+1}, \ldots, m_{0k}\rangle, r_1)$$
$$sym(\texttt{R1 = }\textbf{load}\texttt{ R0,M0}) \triangleq r_1 = \langle read(m_0.1, addr(r_0)), \ldots, read(m_0.j, addr(r_0))\rangle$$

Figure 3.10: Memory-safety aware VCGen semantics.

$$alloc_{sh}(m, r) \triangleq \langle m.val, m.offset, m.size, write(m.alloc, r.base, 1), m.mod\rangle$$
$$free_{sh}(m, r) \triangleq \langle m.val, m.offset, m.size, write(m.alloc, r.base, 0), m.mod\rangle$$
$$store_{sh}(\langle m.alloc, m.mod\rangle, r) \triangleq \langle m.alloc, write(m.mod, r.base, 1)\rangle$$

Figure 3.11: Shadow memory semantics for memory safety.

and shadow memory to track extra data about pointers and memory, respectively. Moreover, we present a general extension of both memory and pointer semantics.

Intuitively, we want to represent each fat pointer as a tuple of values that collectively represent the value of the pointer and all the metadata (i.e., *fat*) that is cached at it. We do not put restrictions on the number of values nor their sorts. However, we assume that there is a function *addr* that maps a pointer to an expression representing an address. Thus, for a pointer register `R`, $sym(\texttt{R})$ is a tuple $\langle t_1, \ldots, t_j\rangle$ of j constants that represents the pointer, and $addr(\langle t_1, \ldots, t_j\rangle)$ is the address of that pointer. For example, a common case is to use the first element of the tuple to represent the address: $addr(\langle t_1, \ldots, t_j\rangle) = t_1$. Fig. 3.13 presents a small program (on the left) that writes a fat pointer `P0` to memory at address `P1`. Memory is divided into five parts with *val* memory used to store the actual program data. Here, *val* stores the *base* value of the fat pointer and *offset* and *size* store the fat. Memory operations are tracked by *alloc* and *mod* memory that mark whether an address is allocated and whether it has been written to, respectively. Fig. 3.13 shows the memory state after the **store** operation. Both *alloc* and *mod* are set to 1 because `P1` is allocated and has been modified.

Formally, we re-define *ptrs* to be a tuple of sorts, written as $\langle s_1, \ldots, s_j\rangle$. We say that a tuple $\tau = \langle c_1, \ldots, c_p\rangle$ of p constants is of a tuple sort $\langle s_1, \ldots, s_p\rangle$ iff, for each $0 < i \leq p$, c_i is of sort s_i. Tuples of sorts, and tuples of constants are only present during VCGen, but not in the final verification condition. For that, we rewrite equality between two tuples as conjunction of equalities between their elements, and use $\tau.i$ for the ith element of tuple τ.

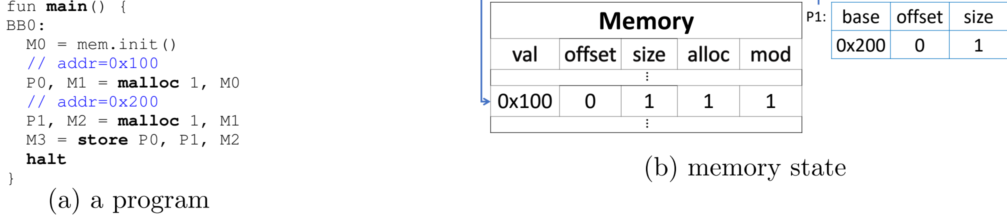
Similar to a pointer register, we re-define *mems* for a memory register `M` as a

$$sym(\texttt{R1} = \textbf{isderef}\ \texttt{R0 B}) \triangleq r_1 = 0 \leq (b + r_0.\mathit{offset}) < r_0.\mathit{size}$$

$$sym(\texttt{R1} = \textbf{isalloc}\ \texttt{R0 M}) \triangleq r_1 = read(m.alloc, r_0.base)$$

$$sym(\texttt{R1} = \textbf{ismod}\ \texttt{R0 M}) \triangleq r_1 = read(m.mod, r_0.base)$$

Figure 3.12: Semantics for verifiying memory safety.



(a) a program

(b) memory state

Figure 3.13: Memory state `M3` - when `P0` is stored at location `P1`.

tuple of values that store program and shadow state . Thus, $sym(\texttt{M}) = \langle v_0, \ldots, v_k \rangle$, where each v_i is the sort $bv(64) \rightarrow bv(64)$. If a pointer is represented by a j-tuple, we assume that memory is represented by a k-tuple, with $k \geq j$, so that the first j entries in a memory register are wide enough to store the fat pointer. Specifically, we require that the sort of v_j is same as sort of t_j for $1 \leq j \leq k$.

We modify the semantics of **`malloc`** by storing meta data along with explicit program states. The modification is defined in Fig. 3.10 (`M1` is now a memory tuple). The signature of *alloc* is unchanged, but now returns a fat pointer. Given a pointer p of sort *ptrs*, a function $size(\langle t_1, \ldots, t_j \rangle)$ returns the size of a memory object pointed-to by p. An additional function $alloc_{sh} : mems \rightarrow mems$ operates on shadow memory. The semantics of $alloc_{sh}$ and $free_{sh}$ is described in Sec. 3.3.2.

A **`store`** is divided into two parts. First is the store of actual program data. Here, since the program data can be of sort *scalr* or *ptrs*, a store of a k tuple of data on memory m_0 is translated into k *write* functions on each element of $\langle m_{01}, \ldots, m_{0j} \rangle$. Second is changing metadata associated with the memory. This is abstracted by the function $store_{sh}$ that works on $\langle m_{0j+1}, \ldots, m_{0k} \rangle$. The specifics of this function is described in Sec. 3.3.2. Similarly a load operation expects to read $\langle m_{01}, \ldots, m_{0j} \rangle$ of sort *ptrs*. The encoding described above is general and allows representing arbitrary fat and shadows. Next, we illustrate specializations for memory safety.

3.3.1 Spatial memory safety

A program satisfies spatial memory safety iff every read and write is always inside an allocated object. A fat pointer is defined as a tuple of three constants $\langle s_1, s_2, s_3 \rangle$ denoted as $\langle \mathit{base}, \mathit{offset}, \mathit{size} \rangle$ for convenience. Here *base* is the start address of the object, *offset* is an index into the object, and *size* is its size. The address *addr* is given by $\mathit{base} + \mathit{offset}$.

With fat pointers, we introduce instructions for pointer arithmetic and pointer integer casts. The instruction **`gep`** is used for pointer arithmetic. Fig. 3.9a shows an example use in `R5 =` **`gep`** `R3, 1`. Here, semantically, a new pointer `R5` is created that has the same *base* and *size* as `R3`, with *offset* incremented by 1. We also introduce **`ptoi`** instruction that casts a pointer to an integer by adding *offset* to *base*. For an integer to pointer cast, we use the **`itop`** instruction. This instruction sets *base* to the integer value and fat to zero.

To assert that a pointer dereference is spatially memory safe, we provide an **`isderef`** instruction, whose semantics is shown in Fig. 3.12. As an example, the program in Fig. 3.9a executes **`assert`**`(0)` as `R6 =` **`isderef`** `R5, 1` evaluates to *false* causing `A1` to evaluate to `R1` and `A2` to evaluate to *true*. This causes the VC in Fig. 3.9b to be *satisfiable* and exposes the out of bounds error in Fig. 3.2 line 9. Note that the same error is not caught by the VC in Sec. 3.2.3. In SeaBMC, we can automatically add **`isderef`** instructions before memory accesses. Many of such assertions are statically and, thus, cheaply resolved to *true* or *false* before SMT solving is invoked.

Note that SeaBMC semantics for spatial safety differs from LLBMC [49]. LLBMC treats only accesses to unallocated memory as unsafe. This implies that it is valid for a pointer to overflow into another object allocated just below or above. In SeaBMC, jumping across the allocated boundary is invalid. SeaBMC also differs from CBMC in this regard. In CBMC [14], the pointer representation is fixed and a few bits in the pointer representation are reserved for fat data. These constrain the available address range. Additionally, only limited metadata can be stored in each pointer. In SeaBMC, we support composite pointer representations that maintain parity with concrete pointer representation while allowing for a rich metadata in the fat region of the pointer.

3.3.2 Temporal memory safety

A program satisfies temporal memory safety iff it never does one of the following: (**UAF**) an object is used after it has been freed; and (**RO**) an object marked as read-only (by programmers) is modified. We detect a violation of memory safety by tracking the status of a memory object using shadow memory. Each memory is a tuple $\langle v_1, \ldots, v_5 \rangle$ of constants of sort $bv(64) \to bv(64)$, denoted $\langle val, offset, size, alloc, mod \rangle$, where $\langle val, offset, size \rangle$ maps to pointer data $\langle base, offset, size \rangle$, and *alloc* and *mod* track the allocated and modified status of an object, respectively.

An object can be in allocated or freed (unallocated) state. To track allocated state, *sym* in Sec. 3.2.3 is extended for **`alloca`**, **`malloc`**, and **`free`**. The new semantics is shown in Fig. 3.10. The function $alloc_{sh} : mems \to mems$ is defined, for temporal memory safety, as shown in Fig. 3.11. Note that $alloc_{sh}(m, r)$ marks $m.mod$ memory only at the start of an object, i.e., $r.base$. For this reason it is necessary to use the fat pointer representation in Sec. 3.3.1 since it records the *base* for every pointer. The **`isalloc`** instruction, shown in Fig. 3.8, is used to check the allocated state of an object at any point in the program. The semantics for **`isalloc`** is defined in Fig. 3.12.

A C program has no native mechanism for verifying that an object remains unmodified when passed to a function. To remedy this, we extend the semantics for **`store`** as shown in Fig. 3.10. The function $store_{sh} : mems \to mems$ is implemented for temporal memory safety as shown in Fig. 3.11. The **`ismod`** in Fig. 3.8 is used to check the read-only state of an object at any program point. The semantics for **`ismod`** is given in Fig. 3.12. We also provide a companion instruction `resetmod R, M` that resets $m.mod$ at address $r.base$ to zero. This allows initializing an object, resetting modified state, and then checking that the subsequent program does not modify the object. We track memory state only at object granularity, therefore, the current implementation is tied to using the fat pointer representation in Sec. 3.3.1.

3.4 Experiments

In this section, we reproduce key results from our companion work [44]. For the full results, please refer to the paper.

We describe the evaluation of SEABMC on verification tasks from `aws-c-common`. Each task targets a single function from `aws-c-common` checking post-conditions

	Statistics		SeaBMC			CBMC			SMACK					KLEE			
category	cnt	loc	avg (s)	std (s)	time (s)	avg (s)	std (s)	time (s)	cnt	fld/to	avg (s)	std (s)	time (s)	cnt	avg (s)	std (s)	time (s)
arithmetic	6	202	1	0	7	4	1	23	6	2/0	7	1	43	6	1	0	**5**
array	4	390	2	1	**8**	6	0	22	4	0/1	58	95	230	4	32	6	129
array_list	23	2 901	5	6	**114**	36	61	824	23	0/0	13	2	303	23	55	49	1 275
byte_buf	29	2 817	2	1	**43**	18	47	512	29	0/2	44	55	1 270	27	75	124	2 034
byte_cursor	24	2 361	2	0	**36**	7	4	165	16	0/2	44	62	702	17	13	14	217
hash_callback	3	347	2	1	**7**	10	6	29	3	0/0	9	4	26	3	64	46	192
hash_iter	4	708	11	16	**43**	13	9	51	4	0/2	153	55	613	3	21	10	62
hash_table	19	3 286	5	7	**99**	24	33	447	19	1/5	72	83	1 358	15	105	333	1 569
linked_list	18	2 082	3	2	48	59	209	1 061	18	0/9	111	94	2 000	18	1	0	**13**
others	2	31	1	0	**2**	4	0	7	1	0/0	4	0	4	1	1	0	1
priority_queue	15	2 817	16	25	**243**	208	303	3 121	15	0/1	29	47	432	15	46	12	695
ring_buffer	6	934	12	10	**75**	20	20	120	6	0/4	137	97	824	6	48	26	289
string	15	1 314	3	1	**44**	6	1	94	15	0/3	48	79	720	15	140	160	2 096
total	168	20 190			**769**			6 476	159	3/29			8 525	153			8 577

Table 3.1: Verification results for SeaBMC, CBMC, SMACK and KLEE. Timeout is 200s. **cnt**, **avg**, **std** and **time**, are the number of verification tasks, average run-time, standard deviation, and total run-time in seconds, per category, respectively (repro. from [44]).

and memory safety. Overall, there are 172 tasks, covering 20K LOC. All verification tasks and detailed results are available at https://github.com/seahorn/verify-c-common. We have chosen these tasks because they represent a real industrial use-case of BMC. We have adapted them from CBMC to be compatible with LLVM-based C verification tools. The details are described in [45] [44].

When comparing SeaBMC to other state-of-the-art bounded analysis tools, we use the *optimal* strategy as described in [44], Section 4. We compare against: CBMC [14], SMACK [47], and KLEE [10]. CBMC is, perhaps, the oldest and most well-known BMC for C programs (not based on LLVM). It is actively used by AWS, and was used for the verification of `aws-c-common`. SMACK is an LLVM-based BMC tool that uses Boogie [39] and Corral [47]. It performed very well on the "SoftwareSystems" category in SV-COMP'21. KLEE is an LLVM-based symbolic execution tool. It does not encode the VC in one shot but rather explores satisfiability of path conditions in a program one path-at-a-time. It is a practical alternative to BMC.

The results are shown in Tab. 3.1. SeaBMC and CBMC solve all verification tasks from `aws-c-common`. SMACK timed out on most instances in its bit-precise mode. It timed out on 29 and failed on 2 in arithmetic mode. KLEE is particularly effective on `linked_list` – showing the benefit of exploring path-at-a-time, when the number of paths is small. Overall, SeaBMC outperforms the competitors on most categories and in the overall run-time. Thus, we conclude that SeaBMC is a highly efficient BMC engine.

3.5 Related Work

Bounded Software Model Checking is a mature program analysis technique. We briefly review only some of the closest related work. Over the years, there have been many model checking tools built on top of the LLVM platform. The closest to ours is the work of Babic [3] and LLBMC [49]. Similarly to [3], we rely on the Gated SSA form to remove all control dependence leaving only data-flows to be represented. However, our encoding is significantly simplified by an intermediate representation that purifies memory flows. Unfortunately, [3] has not been maintained making head-to-head comparison difficult.

We borrow the idea of using lambda-encoding for representing memory from LLBMC [49]. One important advantage of lambdas is that we can represent memory operations such as **`memcpy`** efficiently (while with arrays, these have to be unfolded). In particular, this allows for unbounded verification of loop-free programs that use these operations. The most significant difference from LLBMC is in our encoding of memory safety. In particular, we cache bounds information in the pointer, and check that every access is inside the allocated memory object. In contrast, LLBMC assumes an arbitrary allocator and checks that all accesses are into some allocated memory, not necessarily into the expected object. Unfortunately, there is no public version of LLBMC available, so head-to-head comparison was not possible.

SMACK [24, 47] is probably the most known BMC for LLVM. It is based on Boogie and Corral from Microsoft Research. It is most effective for arithmetic abstraction of software (i.e., abstracting machine integers by arbitrary precision integers). Its model for memory safety relies on complex encoding using universally quantified axioms in Boogie, leading to quantified reasoning in SMT. In contrast, our representation is tuned to perform well with modern SMT solvers. SMACK shares SeaDsa [27, 34] alias analysis with SeaBMC. DIVINE4 [4] is an explicit state model checker that also targets LLVM. However, it uses LLVM 7 while SeaHorn uses LLVM 10. This makes head-to-head comparison difficult. It targets parallel programs, which SeaBMC does not. For sequential programs, it is related to `libFuzzer` and KLEE that we compare with.

3.6 Conclusion

In this chapter, we have presented the techniques behind SeaBMC, a new Bounded Model Checker for C based on LLVM. SeaBMC is path-sensitive, bit-precise, and

provides precise model of memory. It extends traditional memory model with *fat* pointers and *shadow* memory that allow attaching additional data to pointers and memory. We have evaluated SeaBMC against CBMC, SMACK, and KLEE and show significant performance improvements over the competition.

Chapter 4

SeaUrchin: Bounded Model Checking for Rust

4.1 Introduction

Rust is a relatively new systems programming language with similar applications as C and C++. It can run on tiny microcontrollers and high-end servers. Using a strong type system, Rust ensures that successfully compiled programs possess certain safety properties. At the same time, there is no runtime penalty for this safety and Rust programs are as performant as equivalent C and C++ ones. The Rust compiler, `rustc`, uses the LLVM backend to generate executables. With LLVM as a common IR, a natural question is whether SeaHorn and SeaBMC can be used to verify Rust programs. Additionally, since Rust has a strong type system, does it affect the kinds of properties we would like to verify? One promise of Rust's type system is that programs don't suffer from spatial and temporal memory safety bugs[1]. This alone would obviate using BMC for verifying non-functional properties like memory safety. However, the story is not so simple. Rust only guarantees safety for code written in a subset of Rust called Safe Rust. For performance and other reasons, programs can call into Unsafe Rust code where, for example, arbitrary memory reads and writes are allowed. Once this happens, the Rust compiler, `rustc`, cannot verify that safety is maintained and it simply assumes that unsafe code is safe. Thus, even though Rust is safe, there are opportunities for verifying both non-functional and functional properties. This chapter summarizes early experience on applying

[1]See Sec. 3.3.

SeaBMC to Rust using three Research Questions(RQ):

RQ 1: Does LLVM bitcode as IR empower SeaBMC to work for Rust out-of-box? The SeaHorn tool is built to work with LLVM bitcode and when verifying C source, it relies on `clang` turning it into LLVM bitcode. Since `rustc`, the Rust compiler, also targets LLVM bitcode as a backend, it should empower SeaBMC to be used as-is, i.e. without embedding any knowledge of Rust. We explore this hypothesis using a new front-end processing tool called SeaUrchin and show positively that using LLVM bitcode enables verification of Rust code with minimal front-end processing.

RQ 2: Does CaS philosophy apply effectively to Rust? C does not have built-in mechanisms to hide library code from client software. This enables library code to be used effectively to write specifications since all logic to express pre– and post–conditions is available to the user. On the other hand, Rust has strong notions of visibility — all code is private across modules unless explicitly declared public. To study this, we design an experiment to express invariants in Rust standard library components so as to ensure that the components do not panic at runtime. In summary, we find that CaS can be used effectively for Rust programs. However, this can require changing library code to expose previously hidden state.

RQ 3: Can verification become more efficient when using Rust language specific information? Using a common IR like LLVM bitcode removes the engineering burden of writing operational semantics for different source languages. For Rust, it offers the additional benefit that we do not need to build semantics for Safe Rust and Unsafe Rust as these abstractions are removed when compiling into LLVM bitcode. At the same time, a constant push for BMC is to improve efficiency so that it can scale to larger or more complex programs. One way of scaling is to use type information present in Rust programs to simplify the generated verification conditions. Thus, there is a middle ground to explore — where we modify SEA-IR to preserve source language information. This is a long term research direction of SeaUrchin and we explore ideas for future work.

4.2 RQ1: Does LLVM bitcode as IR empower SeaBMC to work for Rust out-of-box?

The Rust compiler, `rustc`, targets LLVM bitcode as its primary backend. SeaHorn (and the SeaBMC engine) work on LLVM bitcode. A natural question is

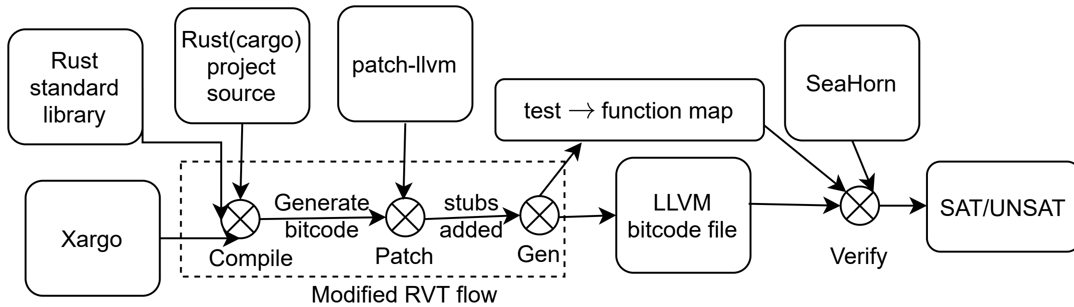


Figure 4.1: Architecture of SeaUrchin.

whether Rust programs can be verified by SeaBMC. To answer this, we implemented a pre-processing pipeline SeaUrchin which converts Rust programs into LLVM bitcode which can be verified by SeaBMC. As shown in Fig. 4.1, SeaUrchin has four stages: **Compile** compiles source (including Rust standard library) into LLVM bitcode; **Patch** replaces computationally expensive code with stubs; and **Gen** generates a source function name to bitcode function name mapping and the writes the patched LLVM bitcode to disk. Finally, the **Verify** stage uses SeaHorn to produce a SAT/UNSAT result.

SeaUrchin, as currently implemented, modifies the Rust Verification Tools (RVT)[2] program for the above steps. RVT is an open-source verification framework from Google Research which enables Rust programs to be verified by different verifiers like SeaHorn and KLEE. It also includes the `patch-llvm` program shown in Fig. 4.1. The modifications to RVT are not very deep but needed for our use case described in Sec. 4.3. For example, RVT (by way of Cargo) does not support compiling the Rust standard library along with a Rust program. However, we need this and we use the Xargo[3] wrapper instead of Cargo.[4]

SeaUrchin is packaged as a startup script `urchin` and a docker image. The script first calls the docker image which contains the modified RVT and then verifies the produced LLVM bitcode using SeaBMC installed locally on a machine. This removes the need for extensive system preparation to run the verification pipeline. The pipeline is open sourced[5]. To put our approach in context, we compare our experience with that of extending SMACK to Rust as documented in [5].

[2] https://github.com/project-oak/rust-verification-tools
[3] https://github.com/japaric/xargo
[4] Newer versions of Cargo obviate the need for Xargo.
[5] https://github.com/priyasiddharth/seaurchin

Extensions to CaS language. Both SMACK and SeaUrchin (through RVT) provide a shallow API to the C verifier.

Generated LLVM bitcode. The work on SMACK [5] describes three differences between LLVM bitcode generated by `clang` and `rustc` and the support added to SMACK to be able to verify Rust code. In our experience SeaBMC worked out of the box for these cases. Namely,

- support for exotic LLVM datatypes like `i1`;
- support for handling compound datatypes packed into primitive datatypes, e.g., a pair `{i32, i32}` packed into a single `i64` integer;
- support for LLVM intrinsics, e.g. `llvm.uadd.with.overflow.i32`.

From our experience in applying SeaBMC to `aws-c-common`, the above LLVM bitcode patterns are produced by sufficiently varied (e.g., industrial) C code and, therefore, SeaBMC was well prepared to handle such code. One similarity with SMACK is that in both cases expensive and unnecessary functionality is stubbed out.

Modifications to Rust standard library. SMACK replaces popular Rust standard library functions with simpler variants for verification. On the other hand, SeaUrchin has the ability to build the Rust standard library from source, with modifications as needed.

Discussion. We implemented SeaUrchin as a preprocessing pipeline for Rust code to show that it is indeed possible to verify a new language out-of-box using a common IR. We also contrast our approach with SMACK. After processing by SeaUrchin, SeaBMC was able to interpret all LLVM bitcode generated by `rustc`. Additionally, we verified Rust projects rather than individual source files like SMACK. We believe this approach allows SeaUrchin to be easily integrated into a development workflow than a single source mechanism would allow. SeaUrchin relies heavily on RVT. The fact that RVT is open source allowed us to make modifications for our use case quite easily. Finally, one advantage of Rust is that it provides a single build mechanism, namely, Cargo. Thus, it is easy to design an integration mechanism for verification tools that works in different contexts. This is useful for industrial software since [6] reports that considerable effort may be spent on integrating static analysis tools with existing build systems in industrial settings.

4.3 RQ2: Does CaS philosophy apply effectively to Rust?

Taking `aws-c-common` as a baseline, it is easy to write specifications using CaS since any data structure could be setup in a desired state as a precondition. Additionally, the postcondition could use *any* state information. This was possible because C has limited mechanisms for data hiding. This is usually considered a hindrance to organize large software systems. However, data visibility is necessary for CaS to be practical since it allows us to write specifications without modifying source code to expose additional state information. Rust has well defined guidelines for data visibility. The Rust language reference[6] says the following:

"By default, everything in Rust is private, with two exceptions: Associated items in a `pub` Trait are public by default; Enum variants in a `pub` enum are also public by default."

4.3.1 Panic freedom in Rust programs

Problem Definition To study how CaS applies to Rust programs, we devised a task for specifying invariants for panic freedom in standard library components. A panic is an error condition which causes a Rust program to execute a panic handler to either correct the error or halt execution. Invariants for panic freedom are useful for a Rust developer to know since maintaining these invariants ensures absence of runtime panics. We specifically target standard library components like `String` and `RefCell`, since they are used widely in practice. Additionally, the invariants for these components are not formally specified but rather are documented informally in prose[7]. For a rapidly evolving language like Rust, it may happen that a developer's understanding of invariants may become stale compared to the implementation. This leads to programs that *may* panic at runtime.

Case Study We first took on the task of expressing invariants for panic freedom for the `String` module. A `String` is the most common type of string with ownership over contents[8]. A `String` causes runtime panic if it is created using invalid `UTF-8` data. For this case study, SeaUrchin wired the panic handler to the `VERIFIER_error` intrinsic. This caused a panic to reach the error state in SeaBMC. Fig. 4.2a shows

[6] https://doc.rust-lang.org/reference/visibility-and-privacy.html
[7] https://doc.rust-lang.org/std/cell/struct.RefCell.html
[8] https://doc.rust-lang.org/std/string/struct.String.html

```
1 #[test]
2 fn test_string_from_bytes_panic() {
3   let b1: u8 = abstract_value();
4   let b2: u8 = abstract_value();
5   let bytes = vec![b1, b2];
6   let v = std::string::String::from_utf8(bytes).unwrap();
7 }
```

(a) String creation with potentially invalid UTF−8 data.

```
1 #[test]
2 fn test_string_from_bytes_nopanic() {
3   let b1: u8 = abstract_value();
4   let b2: u8 = abstract_value();
5   let bytes = vec![b1, b2];
6   assume(
7       std::str::from_utf8(&bytes).is_err() == false);
8   let v = std::string::String::from_utf8(bytes).unwrap();
9 }
```

(b) String creation with valid UTF−8 data.

Figure 4.2: String creation — panic and no-panic versions.

a unit proof which instantiated an invalid `String`. The panic occured in Line 6. The function `abstract_value()` created a nondeterministic value of the requested type. As an example, Line 3 assigns a nondeterministic byte value to `b1`.The function **`assume`** maps to **`assume`** in SeaBMC. Similarly, the **`assert`**`!` macro maps to **`assert`**. This specification language is exported by RVT.

To ensure that `String` creation is panic free, an invariant was added constraining bytes `b1` and `b2` to be valid UTF-8. Fig. 4.2b, Line 7 adds this assumption. Note that the pre-conditions didn't actually ensure that the bytes are UTF-8. We merely constrained the `String` internal function to ensure UTF-8 validity returned no error. Another option was to actually write the explicit preconditions to create valid UTF-8 bytes. This would give stronger guarantees about validity since there is no dependence on an unverified function `str::from_utf8`. We did not take this approach since a simpler one suffices for our demonstration. With the unit proof for `String` creation, it appeared that we can extend CaS from C to Rust quite naturally. The unit proofs looked similar inspite of the cosmetic differences in the specification language. However, this conclusion was premature. We looked at another example `RefCell` where expressing invariants for panic freedom required modifying the code under verification because required state was hidden from the unit proof.

`RefCell` is a container in the Rust standard library for data which validates borrow checking rules at runtime (instead of compile-time), i.e. at each instance during execution the following borrow rules should be invariant (amongst others). The rules are: First `borrow` is never called on a `RefCell` instance when it is already mutably borrowed for writing; and second `borrow_mut` is never called on a `RefCell` instance when it is immutably borrowed for reading. If these rules are violated by a program, panic ensues.

Operationally, the borrowed state is represented by a 32-bit signed integer. A value of 0 means not borrowed. A positive value implies borrowed for reading where the value denotes the number of readers. A value of -1 implies borrowed for writ-

ing. Note that there can only be one writer at a time. This borrowed state is not visible outside the `RefCell` module. We added the following API functions to the `RefCell` implementation in the standard library to set *pre-conditions* and assert *post-conditions*.

1. `cell::is_unused()` – returns `true` if cell is not borrowed for reading or writing, else `false`.

2. `cell::is_reading()` – returns `true` is cell is borrowed for reading, else `false`.

3. `cell::is_writing()` – returns `true` if cell is mutably borrowed for writing, else `false`.

With this we were able to write the unit proof in Fig. 4.3. A new function we use here is `abstract_where`. This serves the same purpose as an **`assume`** – taking a lambda function as an argument. This unit proof shows that a mutable borrow for writing from an unborrowed `RefCell` does not cause a panic, i.e. reality matches expectation.

We then wrote a unit proof in Fig. 4.4a that was intended to verify that adding readers to a `RefCell` which is already in a reading state is valid. Since a `RefCell` can have multiple readers, we expected that the pre-conditions in Line 5 – Line 8 ensured panic freedom. Surprisingly, this was not the case and the test panicked! Looking at the counterexample, this happened because before Line 10, the `RefCell` could have the maximum number of readers that could fit into a signed 32-bit number. Consequently, when another reader was added by adding 1, the borrowed state counter overflowed and rolled over to the maximum negative value possible. The runtime borrow checker then reported an error since the `RefCell` was in the mutably borrowed state (< 0) and therefore could not be borrowed for reading. To fix this, we needed an additional precondition in Fig. 4.4a, Line 5 - Line 8 that assumes that the number of readers is less than the maximum value of a signed 32-bit integer. For this, we added a function `cell::is_reader_limit_reached()` to the `RefCell` API and changed the precondition suitably in Fig. 4.4b.

Discussion We show that it is possible to extend CaS to Rust. In some cases, like `String`, CaS works as it does in C. For others, like `RefCell`, we need to modify the code under verification to expose additional state. This is undesirable from a software engineering standpoint since we override data hiding (which exists for good reasons). To mitigate the problem, we can use the conditional compile feature of `Cargo`. This enable data visibility only for verification and hides it for production

```
1  #[test]
2  fn test_borrow_mut_nopanic() {
3      let val: u32 = abstract_value();
4      let c = RefCell::new(val);
5      let nd_borrow_flag: isize = abstract_where(
6          |x| cell::is_unused(*x));
7      c.set_borrow_state(nd_borrow_flag);
8      let m = c.borrow_mut();
9  }
```

Figure 4.3: unit proof for mutable borrow.

```
1  #[test]
2  fn test_borrow_panic() {
3    let val: u32 = abstract_value();
4    let c = RefCell::new(val);
5    let nd_borrow_flag: isize = abstract_where(|x| {
6        cell::is_unused(*x) ||
7        cell::is_reading(*x)
8      });
9    c.set_borrow_state(nd_borrow_flag);
10   let m = c.borrow();
11 }
```

(a) Borrow for reading causes panic.

```
1  #[test]
2  fn test_borrow_nopanic() {
3    let val: u32 = abstract_value();
4    let c = RefCell::new(val);
5    let nd_borrow_flag: isize = abstract_where(|x| {
6        cell::is_unused(*x) ||
7            (cell::is_reading(*x) &&
8             !cell::is_reader_limit_reached(*x))
9      });
10   c.set_borrow_state(nd_borrow_flag);
11   let m = c.borrow();
12 }
```

(b) Borrow for reading does not cause panic.

Figure 4.4: Borrow for reading - panic and no-panic versions.

use. Additionally, extending the API for verification also enables it to be used for testing. Therefore, such extensions can become a standard way of writing Rust modules enabling both easy testing and verification. Finally, we also note that exposing methods for verification has been explored in the context of JML [9]. Here methods are introduced in JAVA for verification. This can be explored for Rust as well.

4.4 RQ 3: Can verification become more efficient when using Rust language specific information?

A core feature of Rust is *ownership*, a way to record aliases and mutation of data. Ownership is specified by three rules [17]: 1. All values have exactly one owner; 2. A reference to a value cannot outlive the owner; and 3. A value can have one mutable reference or many immutable references.

[9] See https://www.cs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_2.html.

The Rust type system guarantees that once a value has a mutable references, there are no other references (mutable or immutable) to that value. This aliasing information can help generate simpler verification conditions. This, in turn, decreases verification time and enables larger and more complex programs to be verified in a reasonable amount of time. We present our case using two example. In the first example, the Rust type system generates simpler LLVM IR which leads to faster verification. In the second case, the verification conditions generated can be simplified. Both examples use the ownership rule that a value can have only one mutable reference at a time in Rust.

Rust can generate simpler LLVM IR than C We present an example in Fig. 4.5. Here an equivalent program is written in both Rust and C. We first consider the function `bing` in both Fig. 4.5a and Fig. 4.5b. This function takes as a parameter an array of mutable references (pointers in C) to integers. The function then loops through the array updating the memory location pointed to by element. Finally it returns the content of the memory location pointed to by the zeroth element. In the case of a C program, a given location in memory may be pointed to by multiple array elements. For Rust, the fact that the array elements are mutable references ensures that each element points to a unique memory location.

The rest of the program is set up the array similarly for both Rust and C versions. In each case, the array is setup using a nondeterministic striding pattern that does not alias, i.e., in Fig. 4.5b, Line 15 – Line 16, each `arr[i]` points to either `p[2*i]` or `p[2*i + 1]` nondeterministically (similarly for Rust). In the case of C, this complex pattern thwarts the compiler's attempt to reason that there is no aliasing and the compiler generates LLVM IR accounting for aliasing. For Rust, we use a `mut&` type for array elements as shown in Fig. 4.5a, Line 10. Thus if this program is well-formed (i.e. compiles successfully), the array elements are guaranteed to not alias. This reasoning helps generate simpler LLVM IR for Rust. Finally during verification, in the Rust program, reads and writes to all array elements in `bing` are just replaced by a write to `v[0]` and a subsequent read since the **`assert`** is only on that element. In the C program, array writes to all elements have to be simulated during verification. How this complexity affects verification times in shown in Fig. 4.6.

Here, the Rust program is not affected by the size of the array but the C program is. When studying the plot, the following aspects require some interpretation. First, the C program has a considerably faster verification time than the Rust version for small array sizes. This is because Rust programs have a higher preprocessing time in SeaHorn before the SMT solver is hit. The slowness is yet to be root caused but our hypothesis is that `rustc` produces LLVM IR with more abstractions than

```rust
fn bing(v: &mut Vec<&mut usize>) -> usize {
  v.iter_mut().enumerate().for_each(|(idx, e)| {
      **e = idx;
  });
  return *(v[0]);
}

fn main() {
  let sz = 10;
  let mut v: Vec<&mut usize> = Vec::new();
  let mut p: [usize; 2 * sz] = [0; 2 * sz];
  for chunk in p.chunks_mut(2) {
    let idx: usize = verifier::AbstractValue::abstract_where(
        |&x| x == 0 || x == 1);
    v.push(&mut chunk[idx]);
  }
  let r = bing(&mut v);
  verifier::assert!(r == 0);
}
```

(a) Rust program results in *unsat*.

```c
#define MULT_FACTOR 4
#define SIZE 10
int bing(int **v, int size) {
  for (int i = 0; i < size; i++) {
    *(v[i]) = i;
  }
  return *(v[0]);
}

int main() {
  int **v = malloc(sizeof(int *) * SIZE);
  int *payload = malloc(sizeof(int) * SIZE * MULT_FACTOR);
  memset(payload, 0, sizeof(int) * SIZE * MULT_FACTOR);
  for (int i = 0; i < size; i++) {
    int delta = nd_bool() ? 0 : 1;
    v[i] = payload + (2 * i + delta) * sizeof(int);
  }
  int r = bing(v, size);
  sassert(r == 0);
  return 0;
}
```

(b) C program results in *unsat*.

Figure 4.5: An equivalent program in Rust and C to write and read data from an array.

`clang` does – this is expensive for SeaHorn to preprocess as-is. We believe we may be able to engineer more optimized pre-processing for Rust. Second, we see that the Rust program has a steep curve for small array sizes (size=3, size=10). This is because in the Rust standard library, small arrays (vectors) are created on the stack instead of the heap. Incidentally, the logic for small vectors causes a more complex verification condition than the ones on the heap. This, again, requires more analysis. One way to achieve more stable results is to replace the vector module with our own implementation for verification since we have the ability to modify the Rust standard library.

Caching metadata in fat pointers In the above case, we saw how the Rust type system enables simpler VC generation. We look at another way to simplify VC generation by using aliasing information. There are verification scenarios where we want to verify whether a function has operated on some data stored in memory. Fig. 4.7a is a motivating example of this in an extended SEA-IR form. We do not provide a formal definition of the extension since this is still in development. Informally, the IR has been extended to allow arbitrary functions. The functions are *pure* since they explicitly take memory as input and return memory as output.

Here two pointers `P0` and `P1` along with memory `M0` are passed to the function `bong`. The pointers may alias. The function `bong` calls a function `unsafe_fn` in Line 9 and passes argument `P0` and memory `M1`. We want to verify that `unsafe_fn` does not operate on memory pointed to by `P1`. This can happen when `P0` and `P1` alias. We use helper functions (intrinsics) to setup pre- and post-conditions. The intrinsic `sea.rst_chk` sets the given memory at the given pointer argument to 0. The intrinsic

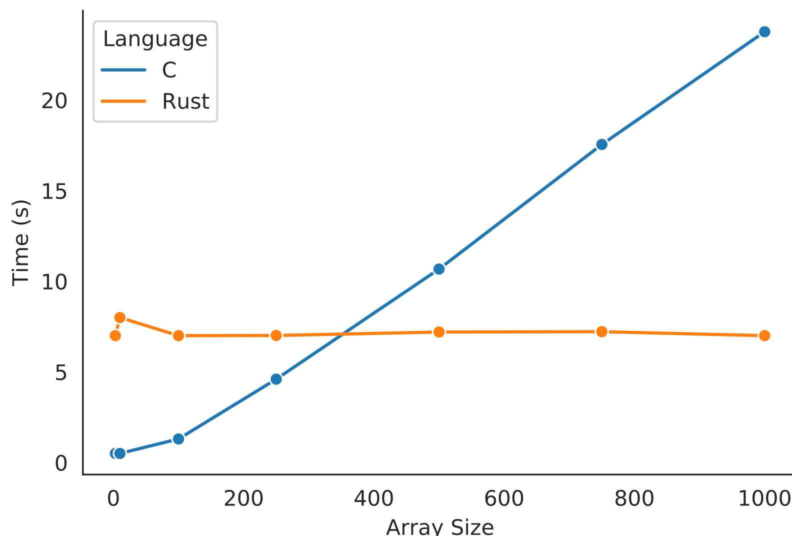


Figure 4.6: Comparision of verification time as array size increases for programs in Fig. 4.5.

`sea.unsafe_fn_invk` returns 1 if the given memory object given by the given pointer was operated using `unsafe_fn`. Fig. 4.7b presents the corresponding VC. We use shadow memory[10]. Shadow memory is given by the array *msha*. When `unsafe_fn` is called, a constraint $msha_2[p_0]$ is 1 is added. A counterexample has a constraint that $msha_2[p_1]$ is 1.

A simpler VC can be generated if aliasing information declares that `P0` and `P1` do not alias, as in Fig. 4.8a. The **`noalias`** keyword in SEA-IR adds this information. The **`noalias`** keyword is present in LLVM IR and generated by the `restrict` keyword in C and mutable references in Rust. When **`noalias`** is found then there is no need to use shadow memory. Instead, we can use fat[11] pointers to record accesses on a memory object. This is shown in Fig. 4.8b. Since `P0` and `P1` do not alias, we can *cache* information in the fat region of pointers denoted by *fat*. Thus, expensive SMT array operations are replaced by constraints on scalar constants.

Discussion The type system of Rust is more expressive and stricter than that of C. This presents unique opportunities for generating simpler VCs leading to faster verification times. In the examples shown above, aliasing information can simplify

[10]See Sec. 3.3.
[11]See Sec. 3.3.

```
1  define bong(P0, P1, M0) {
2  BB0:
3    // reset metadata on memory
4    // pointed to by ptr P1
5    M1 = call sea.rst_chk(P1, M0)
6    M2 = call unsafe_fn(P0, M1)
7    // assert unsafe fn
8    // not called on ptr P1
9    R0 = call sea.unsafe_fn_invk(P1, M2)
10   assert(!R0)
11   ret M2
12 }
```

(a) `P0` and `P1` may alias.

$m_1 = m_0 \wedge msha_1 = msha_0[(p_1 = 0)] \wedge$
$m_2 = m_1 \wedge msha_2 = msha_1[(p_0 = 1)] \wedge$

$r_0 = msha_2[p_1] \wedge$
$r_0 = 1$

(b) Generated VC

Figure 4.7: The function `bong` calls an *unsafe* function.

```
1  define bong(noalias P0, noalias P1, M0) {
2  BB0:
3    // reset metadata on memory
4    // pointed to by ptr P1
5    M1 = call sea.rst_chk(P1, M0)
6    M2 = call unsafe_fn(P0, M1)
7    // assert unsafe fn
8    // not called on ptr P1
9    R0 = call sea.unsafe_fn_invk(P1, M2)
10   assert(!R0)
11   ret M2
12 }
```

(a) `P0` and `P1` do not alias.

$m_1 = m_0 \wedge p_1.fat = 0 \wedge$
$m_2 = m_1 \wedge p_0.fat = 1 \wedge$

$p_1.fat = 1$

(b) Generated VC

Figure 4.8: The **`noalias`** function `bong` calls an *unsafe* function.

VCs in different ways. In fact, optimizations using **`noalias`** information is not exclusive to Rust. C programs which use the `restrict` keyword can also benefit in some cases. The work on exploiting the Rust type system is still in development. Our experiments so far are promising. We hope to work on getting more definitive results by implementing these techniques in SeaBMC and verifying large Rust programs.

4.5 Conclusion

Rust is a higher level programming language than C. It has a strong type system and unique ownership rules. Prima facie, it would seem that the abstractions in Rust would make BMC impractical. However, as shown by the evidence in this chapter, Rust's drive for being as performant as C and C++ has a positive influence on scalability of BMC since many of these abstraction are zero cost. Additionally, the type system results in simpler LLVM IR because the compiler can reason more definitively about a given program. Our extension of SeaBMC to Rust resulted

in the SeaUrchin system. Using SeaUrchin we showed that having a common IR makes adding a new language easy (RQ1). However, to improve verification efficiency, it is desirable to include language specific information (RQ3). We also showed that CaS is generally applicable to a language like Rust. Though, in some cases, deep modifications may be needed to code under verification (RQ2).

Future directions we would like to pursue include:

- scale BMC to work on Rust systems, e.g. a Rust crate, rather than small program;
- automatically infer invariants rather than handcrafting them as in panic freedom example;
- integrate with complimentary verification systems, like the deductive verifier Prusti [1]. Prusti assumes that certain standard library invariants hold. SeaUrchin can assert that the invariants actually hold for a given library implementation.

Chapter 5

Conclusion

In this thesis, we study various aspects of scaling BMC to work on industrial code.

First, we replicate the case study in [11] using SeaBMC and show that the tool is effective an industrial quality library like `aws-c-common`. We further discuss issues around how to write CaS to make it effective across various static and dynamic analysis tools. We also study what kind of bugs exist in code already verified thoroughly using a single tool.

Second, we look at how SeaBMC operates in detail. Our contributions are: an IR, SEA-IR, for LLVM bitcode that purifies memory operations; a VCGen that combines program transformations with encoding into logic allowing for many different styles of VCs; a memory model that combines fat-pointers with shadow-memory to represent metadata; an open-sourced BMC tool; and, a thorough evaluation against the state-of-the-art verification tools on production C code.

Lastly, we extend SeaBMC to work on Rust programs utilizing a common representation format, LLVM bitcode. When expressing invariants for panic freedom, we find that using CaS can be difficult to apply when code to setup and query state is not visible to the *unit proof*. To overcome this, we propose a principled methodology to modify code to expose state for verification only. We also look at using high-level language information in Rust to generate more efficient verification conditions.

To end, we intend to scale BMC in industrial settings. We revisit our definition from Chapter 1. By scaling BMC, we mean three things: 1. large programs; 2. modelling of low-level instructions that occur in practice; and 3. verifying software systems instead of single programs — this may include more than one language. For both `aws-c-common` and Rust programs, we have designed verification for systems

rather than individual programs. For `aws-c-common`, continuous verification builds confidence in the correctness of the library as a whole. For Rust, we verify Cargo projects – this is how Rust code is packaged for distribution. We compile both C and Rust to LLVM IR for verification which ensures that low level instructions are verified rather than a high-level source program. Finally, we verify small pieces of a large system for both `aws-c-common` and Rust programs. BMC is expensive to verify a large software system all at once. Pre- and post-conditions are also manually added to unit proofs. To truly scale, we have to devise techniques that reduce the cost of BMC and automate inferring pre- and post-conditions.

References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30. ACM, 2019.

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. Technical report, ETH Zurich, 2019.

[3] Domagoj Babić. *Exploiting Structure for Scalable Software Verification.* PhD thesis, University of British Columbia, Canada, 2008.

[4] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017.

[5] Marek S. Baranowski, Shaobo He, and Zvonimir Rakamaric. Verifying rust programs with SMACK. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 528–535. Springer, 2018.

[6] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[7] Dirk Beyer. Advances in automatic software verification: SV-COMP 2020. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020,*

Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II, volume 12079 of *Lecture Notes in Computer Science*, pages 347–367. Springer, 2020.

[8] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[9] Marko Kleine Büning, Carsten Sinz, and David Faragó. QPR verify: A static analysis tool for embedded software based on bounded model checking. In Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel, editors, *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*, volume 12549 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2020.

[10] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[11] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Code-level model checking in the software development workflow. In *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, pages 11–20. ACM, 2020.

[12] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In Tibor Gyimóthy, editor, *Compiler Construction, 6th International Conference, CC'96, Linköping, Sweden, April 24-26, 1996, Proceedings*, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1996.

[13] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar

Tasiran, Aaron Tomb, and Eddy Westbrook. Continuous formal verification of amazon s2n. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 430–446. Springer, 2018.

[14] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[15] Byron Cook, Björn Döbel, Daniel Kroening, Norbert Manthey, Martin Pohlack, Elizabeth Polgreen, Michael Tautschnig, and Pawel Wieczorkiewicz. Using model checking tools to triage the severity of security bugs in the Xen hypervisor. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 185–193. IEEE, 2020.

[16] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Model checking boot code from AWS data centers. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 467–486. Springer, 2018.

[17] Will Crichton. The usability of ownership. *CoRR*, abs/2011.06171, 2020.

[18] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *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*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[19] Rob DeLine and Rustan Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, March 2005.

[20] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[21] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2103–2110. ACM, 2010.

[22] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 888–891. ACM, 2018.

[23] Galois. Crux: A Tool for Improving the Assurance of Software Using Symbolic Testing.

[24] Jack J. Garzella, Marek S. Baranowski, Shaobo He, and Zvonimir Rakamaric. Leveraging compiler intermediate representation for multi- and cross-language verification. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, volume 11990 of *Lecture Notes in Computer Science*, pages 90–111. Springer, 2020.

[25] Arie Gurfinkel, Sagar Chaki, and Samir Sapra. Efficient predicate abstraction of program summaries. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2011.

[26] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[27] Arie Gurfinkel and Jorge A. Navas. A context-sensitive memory model for verification of C/C++ programs. In Francesco Ranzato, editor, *Static Analysis*

- *24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, volume 10422 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 2017.

[28] Paul Havlak. Construction of thinned gated single-assignment form. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing, 6th International Workshop, Portland, Oregon, USA, August 12-14, 1993, Proceedings*, volume 768 of *Lecture Notes in Computer Science*, pages 477–499. Springer, 1993.

[29] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 301–306. Springer, 2005.

[30] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In Carla Schlatter Ellis, editor, *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pages 275–288. USENIX, 2002.

[31] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: Build a verifiable filesystem. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, volume 4171 of *Lecture Notes in Computer Science*, pages 49–56. Springer, 2005.

[32] Yunho Kim and Moonzoo Kim. SAT-Based Bounded Software Model Checking for Embedded Software: A Case Study. In *21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*, pages 55–62. IEEE Computer Society, 2014.

[33] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *meltdownattack.com*, 2018.

[34] Jakub Kuderski, Jorge A. Navas, and Arie Gurfinkel. Unification-based pointer analysis without oversharing. In Clark W. Barrett and Jin Yang, editors, *2019*

Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019, pages 37–45. IEEE, 2019.

[35] Orna Kupferman. Sanity checks in formal verification. In *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2006.

[36] Akash Lal and Shaz Qadeer. Powering the static driver verifier using Corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 202–212. ACM, 2014.

[37] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[38] Chris Lattner and Vikram S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 129–142. ACM, 2005.

[39] K. Rustan M. Leino. This is Boogie 2, 2008.

[40] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 1–15. ACM, 2016.

[41] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, volume 7152 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2012.

[42] Yannick Moy and Angela Wallenburg. Tokeneer: Beyond formal program verification. *Embedded Real Time Software and Systems*, 24, 2010.

[43] Roy Osherove. *The Art of Unit Testing: With Examples in .Net.* Manning Publications Co., 2009.

[44] Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, and Arie Gurfinkel. Bounded model checking for llvm, 2021. Submitted to SEFM 2021.

[45] Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, and Arie Gurfinkel. Verifying verified code. In *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Proceedings*, Lecture Notes in Computer Science. Springer, 2021.

[46] Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, and Arie Gurfinkel. Verifying Verified Code. submitted, 2021.

[47] Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014.

[48] Kostya Serebryany. libFuzzer: A library for coverage-guided fuzz testing.

[49] Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *5th International Workshop on Systems Software Verification, SSV'10, Vancouver, BC, Canada, October 6-7, 2010.* USENIX Association, 2010.