CBMC-SSM: Bounded Model Checking of C Programs with Symbolic Shadow Memory

Bernd Fischer
Stellenbosch University
Stellenbosch, South Africa
bfischer@sun.ac.za

Salvatore La Torre
University of Salerno
Fisciano, Italy
slatorre@unisa.it

Gennaro Parlato
University of Molise
Pesche, Italy
gennaro.parlato@unimol.it

Peter Schrammel
University of Sussex and Diffblue Ltd
Brighton and Oxford, UK
p.schrammel@sussex.ac.uk

ABSTRACT

Dynamic program analysis tools such as Eraser, TaintCheck, or ThreadSanitizer abstract the contents of individual memory locations and store the abstraction results in a separate data structure called *shadow memory*. They then use this meta-information to efficiently implement the actual analyses. In this paper, we describe the implementation of an efficient symbolic shadow memory extension for the CBMC bounded model checker that can be accessed through an API, and sketch its use in the design of a new data race analyzer that is implemented by a code-to-code translation.
Artifact/tool URL: https://doi.org/10.5281/zenodo.7026604
Video URL: https://youtu.be/pqlbyiY5BLU

ACM Reference Format:
Bernd Fischer, Salvatore La Torre, Gennaro Parlato, and Peter Schrammel. 2022. CBMC-SSM: Bounded Model Checking of C Programs with Symbolic Shadow Memory. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3551349.3559523

1 INTRODUCTION

Program analysis tools for memory-related properties track meta-information for each memory location and update this as they analyze the unit under test (UUT). For example, Eraser [9] tracks the locks held when each memory location is accessed, while TaintCheck [7] tracks which memory locations contain values that are derived from untrusted sources.

Dynamic analysis tools often store the meta-information in a *shadow memory*, a data structure that is invisible to the UUT but can be efficiently accessed by the tool through the UUT's (original) memory addresses. For example, Eraser's "half-and-half" implementation confines the UUT to the lower half of the system's memory while the shadow memory occupies the upper half; the access to the shadow memory then simply flips the top bit of any given address.

The shadow memory concept simplifies many dynamic program analyses. We believe that it can also be useful for static program analyses, in particular for SAT-based *bounded model checking* (BMC) tools, but it has not been widely explored there yet. Here we describe and demonstrate CBMC-SSM, a *symbolic shadow memory* extension of CBMC [3]. More specifically, we introduce in Section 2 a symbolic shadow memory API that allows us to transparently manipulate shadow variables directly within the input C program, while keeping overheads in the generated SAT formulas low.

The shadow memory extension is particularly useful for designers and developers of program analysis and verification tools, because it enables them to make CBMC handle a wider range of properties over sequential and concurrent programs without having to modify CBMC. Effectively, this enables them to turn CBMC into a programmable program analysis machine; however, the approach is not fundamentally limited to CBMC but applies to other BMC tools as well, e.g., ESBMC [4]. The implementation of a new analysis with CBMC-SSM follows a generic two-stage approach. In the first stage, developers implement the analysis as a code-to-code translation, but use the API to maintain the analysis results in the shadow memory. In the second stage, CBMC-SSM is used to analyze this program version. We illustrate this approach in Section 3 with a simple taint analysis example.

We have used CBMC-SSM to implement a full-fledged data race analysis for C-programs. The implementation builds on our Lazy-CSeq transformation framework [5, 6] and requires only about 600 lines of Python code (see Section 5 for more details). In the SV-COMP 2022 data race benchmarks, it found true races in 66 of the 76 benchmarks with races, and correctly identified 70 of the 78 race-free benchmarks, with only two false negatives and no false alarms, and outperformed custom-developed static data race detectors such as ThreadSanitizer [11] and POR-SE [10].

2 A SYMBOLIC SHADOW MEMORY API

The design of the symbolic shadow memory needs to balance efficiency and simplicity. At the one extreme, we can implement and "seal" it inside CBMC, and implement the custom analysis as a CBMC extension that directly works with this sealed implementation. This means that we effectively implement a one-off custom variant of CBMC that can be used as a black box to analyze unchanged programs. The tight integration of CBMC, shadow memory, and custom analysis ensures efficiency, but CBMC's complex

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3559523

architecture makes the implementation cumbersome and difficult to adapt or reuse for other analyses.

At the other extreme, we can represent the shadow memory explicitly at the source level, through additional program variables, and formulate the custom analysis as a code-to-code translation that inserts code to declare and update the program variables representing the shadow memory. We can then use CBMC as an off-the-shelf symbolic analysis tool to analyze the instrumented program. This separation of concerns simplifies the implementation and increases reusability, but the explicit representation of the shadow memory at the source level increases the size and complexity of both the program and the generated SAT formulas.

We can, however, get both efficiency and simplicity if we maintain the meta-level nature of the shadow memory and bypass its handling by CBMC, but allow the programmatic manipulation of the shadow memory at the source level. This is the purpose of the shadow memory API we introduce here. Its basic idea is that an analysis declares the shadow memory *fields* of the types that it needs, and then uses a single generic getter/setter pair to access and update these fields. Since we assume that the API is used primarily by program transformations implementing custom analyses, we do not provide many convenience methods; instead we keep the API to the following four primitives.

void __CPROVER_field_decl_global(const char* *name*, char *init*) and void __CPROVER_field_decl_local(const char* *name*, char *init*), respectively, declare a shadow memory field with the given *name* and initial value *init*, with the type given by a cast as part of the expression *init*, that is allocated for each *global* (i.e., global variable, static variable, or dynamically allocated memory block) and *local* (i.e., local variable or parameter) memory object, respectively.

char __CPROVER_get_field(void* *address*, const char* *name*) returns the value of the shadow memory field *name* associated with the memory object at the given *address*. The returned value can be cast into the type given by the corresponding shadow field declaration call.

void __CPROVER_set_field(void* *address*, const char* *name*, char *value*) sets the value of the shadow memory field *name* associated with the memory object at the given *address*. *value* is cast to the type given by the corresponding shadow field declaration call.

The split into *global* and *local* scopes allows us to define different fields for different classes of memory objects. This is useful for analyses where only global objects need to be tracked (e.g., data race analysis). Note, however, that information flow analyses (e.g., taint analysis) require all objects to be tracked, and thus require two declarations with the same field name (see Fig. 1, lines 46–47).

3 ILLUSTRATIVE EXAMPLE

In Figure 1, we show how we can use the shadow memory API for a simple taint analysis. The example code constructs an untainted (uname) and a tainted (passwd) string (lines 20–27 and 50–51), packs them into a JSON-formatted composite string (lines 12–19), and then checks that the taint has not infected the untainted parts of the composite string (lines 28–43). Note that simply tracking the taint status of the entire string will yield a false alarm because it is composed from tainted and untainted sources.

```
 1 extern int nondet_int();
 2 int append(char *buf, int pos, char *src){
 3   int len = strlen(src);
 4   for(int i=0; i<len; ++i) {
 5     buf[pos + i] = src[i];
 6     // propagate taint
 7     __CPROVER_set_field(&buf[pos + i], "tainted",
 8               __CPROVER_get_field(&src[i], "tainted"));
 9   }
10   return pos + len;
11 }
12 void encode(char *buf, char *uname, char *passwd){
13   int pos = append(buf, 0, "{\"username\":\"");
14   pos = append(buf, pos, uname);
15   pos = append(buf, pos, "\",\"password\":\"");
16   pos = append(buf, pos, passwd);
17   pos = append(buf, pos, "\"}");
18   buf[pos] = '\0';
19 }
20 void make_nondet_len_string(char *buf, _Bool taint){
21   int len = nondet_int();
22   __CPROVER_assume(0 <= len && len < 8);
23   buf[len] = '\0';
24   // taint the input data
25   for(int i=0; i<len; ++i)
26     __CPROVER_set_field(&buf[i], "tainted", taint);
27 }
28 int check_part(char *buf, int pos, char *src, _Bool expected){
29   int len = strlen(src);
30   for(int i=0; i<len; ++i) {
31     _Bool actual = __CPROVER_get_field(&buf[pos + i], "tainted");
32     assert(actual == expected);
33   }
34   return pos + len;
35 }
36 void check(char *buf, char *uname, char *passwd){
37   int pos = check_part(buf, 0, "{\"username\":\"", 0);
38   pos = check_part(buf, pos, uname, 0);
39   pos = check_part(buf, pos, "\",\"password\":\"", 0);
40   pos = check_part(buf, pos, passwd, 1);
41   pos = check_part(buf, pos, "\"}", 0);
42   check_part(buf, pos, "\0", 0);
43 }
44 void main(){
45   // declare shadow fields
46   __CPROVER_field_decl_local("tainted", (_Bool)0);
47   __CPROVER_field_decl_global("tainted", (_Bool)0);
48   // create harness for SUT
49   char uname[8]; char passwd[8];
50   make_nondet_len_string(uname, 0); // untainted
51   make_nondet_len_string(passwd, 1); // tainted
52   // call SUT
53   char json[46];
54   encode(json, uname, passwd);
55   // check properties
56   check(json, uname, passwd);
57 }
```

Figure 1: Taint analysis using shadow memory API.

For the analysis, we declare a shadow field "tainted" that tracks the taint status of each allocated memory location in a single zero-initialized bit (lines 46–47). These bits are updated via the __CPROVER_set_field method whenever the shadowed objects are updated (lines 7–8). Note that in order to simplify the presentation here we only show the updates for the shadow fields attached to the character arrays; a full taint analysis must also add updates for all local variable writes. However, since the shadow memory declarations and operations are inserted automatically by a source-to-source transformation implementing the taint analysis, this imposes no specification burden on the user.

Note also that __CPROVER_set_field and __CPROVER_get_field access the shadow memory via the addresses of the shadowed objects. This enables CBMC to precisely track the taint status of each individual object in the presence of aliasing, pointers, and complex array indexing operations (see for example line 7).

4 CBMC-SSM IMPLEMENTATION

Shadow memory model C's low-level memory model that allows aliasing and pointer arithmetics requires that the shadow memory is kept separate from (rather than integrated with) the object memory, in order to maintain CBMC's precision. For efficiency reasons, the structure of the shadow memory should be aligned with CBMC's underlying memory model. CBMC employs a heterogeneous, object-based memory model, where symbolic addresses are represented as pairs (object identifier, offset within object)—as opposed to for example an untyped, array-based memory model, where addresses are non-deterministically chosen integers [12]. We allocate for each declared field and each shadowed object a shadow object of the same size and structure, and thus get the same offsets to address the elements within that block of memory. This both simplifies and optimizes mapping a pointer into a shadowed object (i.e., the symbolic address of a shadowed object and the symbolic offset within that object) to a corresponding pointer into the shadow memory object since CBMC-SSM can use CBMC's code to compute the object identifiers as well as the symbolic (i.e., SAT-level) representation of the corresponding offset.

In order to fully account for byte-level aliasing (e.g., a write into a byte array aliased with an integer, which taints an individual byte in the integer), CBMC-SSM replicates the shadow memory for each individual byte of the shadowed object and implements a scatter-gather style access: the setter writes into all replicated copies while the getter returns their aggregated values.

As consequence of this approach, the size of the shadow fields cannot be larger than that of the shadowed objects; in order to simplify the access code into the shadow memory, we even restrict the size of the shadow fields to a single byte. As further consequence, we support individual shadowing only for byte-addressable objects.

Declaration The API functions __CPROVER_field_decl_global and __CPROVER_field_decl_local define shadow memory fields that are allocated for each respective memory object. The second parameter specifies the initial value, but indirectly also determines the type of the shadow memory field, which is required by CBMC to produce the correct SAT-formula. Our implementation currently supports signed or unsigned types with up to eight bits width, e.g.,

```
__CPROVER_field_decl_global("enabled", (_Bool)0);
__CPROVER_field_decl_global("counter", (char)0);
__CPROVER_field_decl_global(
    "stat", (unsigned __CPROVER_bitvector[3])0);
```

where __CPROVER_bitvector[*n*] is an *n*-bit primitive data type provided by CBMC. All field declarations must appear at the beginning of the entry point of the main function.

Setter The API function void __CPROVER_set_field sets the value of the shadow memory object addressed by the given pointer (of type *base_type**) for the given field. If the pointer cannot be dereferenced (i.e., it is NULL or statically determined to be invalid) then the statement will be ignored with a warning. The shadow memory will be set to value for all sizeof(*base_type*) bytes starting from the address given by the value of the pointer. CBMC-SSM aborts if the field has not been declared, or if *base_type* is void or an array type. For example, assuming x and s.sub.y are both of type int, the following calls succeed:

```
__CPROVER_set_field(&x, "enabled", 1);
__CPROVER_set_field(&(s.sub.y), "counter", 5);
__CPROVER_set_field(&(s.sub), "stat", 3);
```

Getter The API function __CPROVER_get_field returns the value of the shadow memory object addressed by the given pointer (of type *base_type**) for the given field. CBMC aborts if the field has not been declared, or if *base_type* is void. Also note that *base_type* cannot be an array type (as types for pointers to arrays or scalars cannot be syntactically distinguished in C) but can be a struct or union containing an array member.

If the pointer cannot be dereferenced the field's declared initial value is returned. If the pointer is valid then the return value is composed from the sizeof(*base_type*) bytes starting from the address given by the value of the pointer. The composition operations are *logical or* for shadow fields of type _Bool and *max* for other types. For example, if x is a base type and enabled is declared as above, the return value enabled of the call

```
_Bool x_enabled = (_Bool)__CPROVER_get_field(&x, "enabled");
```

is the *logical or* over the values stored in the shadow memory bytes corresponding to the bytes of x. Likewise, assuming s.sub is of type struct sub_data, the return value s_sub_stat of the call

```
unsigned __CPROVER_bitvector[3] s_sub_stat =
    (unsigned __CPROVER_bitvector[3])
    __CPROVER_get_field(&(s.sub), "stat");
```

is the maximum of the values stored in the shadow memory bytes corresponding to the bytes of s.sub.

This composition handles aliasing, where parts of the shadow memory can be modified through a different address, and propagates shadow memory updates from individual members to their enclosing (sub-) structures. Hence, we can retrieve the shadow value by accessing any of the constituting bytes, which is needed for a reasonable shadow memory semantics of unions and bitfields.

Symbolic Execution The shadow memory API calls are interpreted by the symbolic execution phase of CBMC, which unwinds the program's control flow graph a bounded number of times. This results in a single static assignment representation. This is then encoded into a propositional formula by bitblasting such that the formula is satisfiable if and only if an assertion in the program fails. This is determined by passing the formula to a SAT solver.

Usage CBMC-SSM is a drop-in replacement for CBMC and takes the same command-line arguments as the original CBMC. It automatically handles all calls to the shadow memory API described earlier. We can thus analyze our taint analysis example in Fig. 1 using the command `cbmc-ssm example.c --unwind 15`, assuming that the code in Fig. 1 is in file example.c. The longest loop in the example requires 15 iterations, thus we can safely bound loop unwinding with --unwind 15. CBMC reports `VERIFICATION SUCCESSFUL` within a fraction of a second, which shows that the code does not assign any tainted values to untainted variables.

5 DATA-RACE DETECTION USING CBMC-SSM

We used the CBMC-SSM API to design and implement, in less than four weeks, a full-fledged data race analyzer for multi-threaded C programs. This finds data races under an interleaving semantics, i.e., when two threads access the same shared memory location immediately subsequently, the accesses can non-deterministically be executed in either order, and at least one access is a write operation.

This approach only requires a shadow bit for each shared memory location and a few auxiliary flags. More specifically, we non-deterministically select an arbitrary interleaving where the last statement of a context C_1 writes to an arbitrary shared location ℓ, and record in the shadow memory for ℓ that this location represents the "write part" of the data race, by setting the shadow flag to true. We then confirm the data race by checking via the shadow memory that the first statement of the interleaving's next (non-empty) context C_2 also accesses the same location ℓ.

This combination of shadow memory and non-determinism (which allows us to "pick" the right variable and context switches) frees us from having to encode more complex deterministic data structures at the SAT formula level. This in turn offsets the costs of the injected updates to the shadow memory and control flags and results in a scalable solution. In particular, our prototype implementation found true races in 66 of the 76 benchmarks with races from the SV-COMP 2022 data race benchmarks, and correctly identified 70 of the 78 race-free benchmarks, with only two false negatives and no false alarms, and outperformed custom-developed static data race detectors such as ThreadSanitizer and POR-SE.

We leveraged the Lazy-CSeq transformation framework [5, 6] to inject the control code that manipulates the shadow memory and auxiliary flags. This allowed us to quickly test out several design alternatives, and made the code very compact—the implementation handles the entire C-syntax in about 600 lines of Python code.

6 RELATED WORK

With the dramatic improvements in SAT solving, SAT-based BMC tools (e.g., CBMC or ESBMC [4]) and symbolic execution engines (e.g., Klee [2]) have become widely used software engineering workhorses. However, very few of these tools support a shadow memory. UC-Klee [8] is a Klee extension that uses the shadow memory to track whether each symbolic input byte is under-constrained or fully constrained, similar to the taint analysis sketched in Section 3. Unfortunately, it does not provide an API but seals the implementation inside Klee, and thus remains difficult to extend to other applications. Without an API to programmatically manipulate the lower-level formula representation, developers typically resort to writing an interpreter for their new analysis at the source code level (e.g., the sequentialization techniques for weak memory models [13, 14]), making the underlying tool a meta-interpreter. We believe that symbolic shadow memory API enables a more systematic and more efficient approach to extending such tools. Sys [1] provides a very rich API in form of a complex domain-specific language and has been used to build a number of specialized symbolic analyses. However, it is based on a flat memory model, where as the heterogeneous memory model employed by CBMC and top-performing BMC tools makes an integration more difficult and favors a simpler API as the one we propose here.

7 CONCLUSIONS

The notion of shadow memory has found wide use as a design pattern for building dynamic program analysis tools. In this paper, we transferred it to SAT-based BMC tools, and introduced the complementary notion of *symbolic shadow memory* that allows a user program to transparently manipulate shadow variables directly within the input C program, while keeping overheads in the generated SAT formulas low. We extended CBMC with a symbolic shadow memory, and discussed how this can be used to perform taint analysis and data race detection with little effort.

We believe that the symbolic shadow memory concept introduced here enables a generic approach to leveraging the impressive capabilities of existing SAT-based bounded program analysis tools to quickly create new symbolic analysis tools for a wide range of properties over sequential and concurrent programs—effectively, it allows us to turn tools like CBMC into programmable program analysis machines.

We plan to explore this new tool design space in future work. We have already prototyped a range abstraction, a weak memory model verifier based on sequentialization, and partial order reductions for symbolic explorations. We believe that this can also be used.

Acknowledgments This work was partially supported by INDAM-GNCS 2020-2022 and 2021 Visiting Professor grants, and by AWS 2021 Amazon Research Awards.

REFERENCES

[1] F. Brown, D. Stefan, and D. R. Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. *USENIX Security Symp.*, pp. 199–216. USENIX Assoc., 2020.

[2] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI*, pp. 209–224. USENIX Assoc., 2008.

[3] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *TACAS*, LNCS 2988, pp. 168–176, 2004.

[4] L. C. Cordeiro, B. Fischer, and J. Marques-Silva. Smt-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.

[5] O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded C-programs. *ASE*, pp. 807–812. IEEE Comp. Soc., 2015.

[6] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. *CAV*, LNCS 8559, pp. 585–602. Springer, 2014.

[7] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *NDSS*. The Internet Society, 2005.

[8] D. A. Ramos and D. R. Engler. Under-constrained symbolic execution: Correctness checking for real code. *USENIX Security Symp.*, pp. 49–64. USENIX Assoc., 2015.

[9] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[10] D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, and K. Wehrle. Symbolic partial-order execution for testing multi-threaded programs. *CAV*, LNCS 12224, pp. 376–400. Springer, 2020.

[11] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with LLVM compiler - compile-time instrumentation for ThreadSanitizer. *Runtime Verification*, LNCS 7168, pp. 110–114. Springer, 2011.

[12] C. Sinz, S. Falke, and F. Merz. A precise memory model for low-level bounded model checking. *Worksh. Systems Software Verification*. USENIX Assoc., 2010.

[13] E. Tomasco, T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Lazy sequentialization for TSO and PSO via shared memory abstractions. *FMCAD*, pp. 193–200. IEEE, 2016.

[14] E. Tomasco, T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Using shared memory abstractions to design eager sequentializations for weak memory models. *SEFM*, LNCS 10469, pp. 185–202. Springer, 2017.