ESBMC-CHERI: Towards Verification of C Programs for CHERI Platforms with ESBMC

Franz Brauße
The University of Manchester, UK

Fedor Shmarov
The University of Manchester, UK

Rafael Menezes
The University of Manchester, UK

Mikhail R. Gadelha
Igalia, A Coruña, Spain

Konstantin Korovin
The University of Manchester, UK

Giles Reger
The University of Manchester, UK

Lucas C. Cordeiro
The University of Manchester, UK
lucas.cordeiro@manchester.ac.uk

ABSTRACT

This paper presents ESBMC-CHERI – the first bounded model checker capable of formally verifying C programs for CHERI-enabled platforms. CHERI provides run-time protection for the memory-unsafe programming languages such as C/C++ at the hardware level. At the same time, it introduces new semantics to C programs, making some safe C programs cause hardware exceptions on CHERI-extended platforms. Hence, it is crucial to detect memory safety violations and compatibility issues ahead of compilation. However, there are no current verification tools for reasoning over CHERI-C programs. We demonstrate the work undertaken towards implementing support for CHERI-C in our state-of-the-art bounded model checker ESBMC and the plans for future work and extensive evaluation of ESBMC-CHERI. The ESBMC-CHERI demonstration and the source code are available at https://github.com/esbmc/esbmc/tree/cheri-clang.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**.

KEYWORDS

bounded model checking, formal methods, capability hardware, CHERI, ARM Morello

ACM Reference Format:
Franz Brauße, Fedor Shmarov, Rafael Menezes, Mikhail R. Gadelha, Konstantin Korovin, Giles Reger, and Lucas C. Cordeiro. 2022. ESBMC-CHERI: Towards Verification of C Programs for CHERI Platforms with ESBMC. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3533767.3543289

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07...$15.00
https://doi.org/10.1145/3533767.3543289

1 INTRODUCTION

Memory safety issues remain the main (~70%) source of security vulnerabilities [10] in software. To combat these vulnerabilities in legacy systems, the Digital Security by Design (DSbD) project[1] has introduced a new hardware-based protection architecture called CHERI [20]. It provides runtime protection (via raising hardware interrupts on unsafe behavior) that prevents such vulnerabilities from being exploited. This setup means that we still want to remove memory safety vulnerabilities from code but are protected if we do not. One successful technique for detecting memory safety vulnerabilities is bounded model checking [2]. However, such tools cannot be immediately applied to code compiled for CHERI as this architecture modifies the runtime system. This paper describes our efforts to extend the ESBMC [5, 7] model checker so that it can be applied directly to C code targeting the CHERI platform.

Capability Hardware Enhanced RISC Instructions (CHERI) provides an extended set of instructions for RISC architecture [23]. The CHERI model implements memory access via hardware capabilities - tokens restricting access to a particular region of the virtual address space. These capabilities are used to directly extend C/C++ with so-called 'fat pointers' (referred to as *capabilities*) for providing memory protection at the hardware level [20]. From being a theoretical and a software-emulated technology, CHERI has now been realised as the ARM Morello[2] system-on-a-chip development board featuring a CHERI-extended ARMv8-A processor.

DSbD is a £187m project with a cross-cutting collection of industrial and research partners. Recently, £7.9m was invested in enriching the software ecosystem for ARM Morello hardware[3] targeting many ambitious projects (e.g., adapting a complete desktop environment containing over 60 million lines of code to Morello [19]). Even with this growing industrial user base, until now, there have been no verification tools available for formal reasoning about C programs written for capability platforms. Furthermore, existing tools cannot directly handle such programs as CHERI capabilities introduce new semantics to C programs (explained below). Hence, it is essential to be proactive in developing verification tools for capability C code to cope with the upcoming demand.

Although CHERI capabilities provide memory safety at the hardware level, they establish the last line of defense. In other words,

[1] https://www.dsbd.tech/about/
[2] https://www.arm.com/architecture/cpu/morello
[3] https://www.ukri.org/news/developing-a-software-ecosystem-for-a-more-secure-digital-future/

they ensure the program crashes safely if a safety violation is detected. Such crashes should still be avoided in practice. At the same time, the runtime protection introduced by CHERI reduces the set of allowed executions, i.e., there are safe C programs that produce hardware exceptions when compiled as CHERI-C programs, which poses a software portability issue between the platforms. Moreover, CHERI capabilities do not yet protect against some known classes of vulnerabilities. For instance, the support for temporal memory safety (e.g., use after free, dangling pointers) is implemented at a software level (as an extension of the CheriBSD virtual-memory subsystem) [21]. Detecting the above issues statically is crucial for preventing the unsafe and/or undesired behavior in the programs running on capability hardware.

We are addressing this problem by implementing ESBMC-CHERI – an extension that enables CHERI-C support in our state-of-the-art model checker ESBMC [5, 7]. ESBMC is a powerful context-bounded model checker for verifying single- and multi-threaded C/C++ programs for various code safety violations (e.g., buffer overflows, dangling pointers, arithmetic overflows) and user-defined assertions. ESBMC has found bugs in real-world software (e.g., libSPDM , sniffer application [12], embedded software from NEC Corporation [5]). ESBMC's modular structure allows relatively easy adaptations of new programming languages [16, 11]. Moreover, ESBMC has been achieving high positions in SV-COMP[4] and Test-COMP[5] – the two major software verification and testing competitions for programs – accumulating over 25 awards over the past 10 years.

In ESBMC-CHERI, we lay down the groundwork for the incremental extension of ESBMC towards verifying C programs for capability platforms [23]. Namely, we introduce a capability-aware memory model into ESBMC, integrate the CHERI clang front-end, and implement a computational model for CHERI-C API.

2 CHERI MODEL

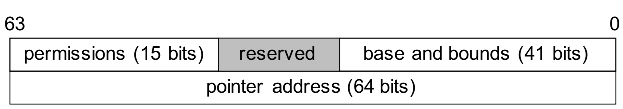

Figure 1: CHERI 128-bit compressed capability.

CHERI introduces architectural capabilities encoding information such as the pointer's address, the object size, and the base address. Furthermore, unlike traditional fat pointers, they store information about the access permissions for the addressed region and some additional metadata. This information is used for various checks (e.g., the pointer's bounds, the permission rights) at the hardware level upon every memory access. A hardware exception is triggered if a violation is detected. In practice, CHERI uses compressed capabilities to reduce the performance overheads and the increased memory consumption. Thus, CHERI capabilities are compressed down to double the size of machine word size (i.e., 128 bits for 64-bit platforms) [22]. However, due to the applied compression, not all memory bounds can be precisely encoded for larger allocations which means that some safe (in the context of non-capability platforms) C programs might produce hardware exceptions on CHERI platforms. Figure 1 demonstrates the in-memory representation of a 128-bit CHERI capability. Additionally, each capability is associated with a validity tag – a single unforgeable bit living in unaddressable memory (i.e., tagged memory) and determining whether the capability is derived from an addressable memory location.

CHERI provides support to two different capability models: a pure-capability and a hybrid model [20]. The former treats all C/C++ pointers as capabilities. At the same time, the latter allows the coexistence of capabilities and regular pointers via additional compiler infrastructure that can be used to specify capability operations, which are translated into the CHERI extended ISA instructions [23]. In addition, CHERI platforms provide a C API to manipulate and query capabilities (e.g., `cheri_setbounds(c,n)` derives from `c` a new capability allowing access only to the `n` bytes starting at `c`'s address, if this is a subset of `c`'s permissions). Listing 1 shows a hybrid CHERI-C program, which will result in a hardware exception during execution on a CHERI platform as `b-1` evaluates to a capability pointing outside of `b`'s original bounds. This means that an actual execution never reaches line 8. ESBMC-CHERI finds this safety violation and reports an assertion failure in line 6. More hybrid CHERI-C examples supported by ESBMC-CHERI are available at the GitHub repository.

Listing 1: CHERI-C code example

```
1 #include <cheri/cheric.h>
2 void main(void) {
3     int n = nondet_uint() % 1024; // models user input
4     char a[n+1], *__capability b = cheri_ptr(a, n+1);
5     b[n] = 17;          // succeeds
6     char *__capability c = cheri_setbounds(b-1, n);
7     /* ... */
8     memset_c(c, 42, n); // like memset() for capabilities
9 }
```

3 ESBMC ARCHITECTURE

ESBMC transforms a given C program using a Clang-based [8] front-end into an intermediate representation in the GOTO language [4], which is symbolically executed to produce logical formulae (see Figure 2). These formulae are passed to one or more specified SMT solvers producing the verification result: either all properties hold in the given program up to the given execution depth k, or one of the properties has been violated.

ESBMC can identify various spatial and temporal memory safety violations (e.g., buffer overflows, dangling pointers, memory leaks) in C programs and verify user-defined assertions. Moreover, ESBMC provides computational models (i.e., "approximations" implemented in C and/or using ESBMC intrinsic functions) for external libraries if their C source code is unavailable. For example, ESBMC models standard C library functions such as *memset, memcpy, malloc, free*.

Table 1 presents the extensions (with their corresponding implementation progress) we identified to be essential for enabling hybrid CHERI-C support in ESBMC. They can be briefly divided into three independent packages (highlighted in Figure 2): 1) integrating the CHERI-clang compiler into ESBMC's front-end, 2) extending the ESBMC memory model with CHERI capabilities and

[4]https://ssvlab.github.io/esbmc/sv-comp.html
[5]https://ssvlab.github.io/esbmc/test-comp.html

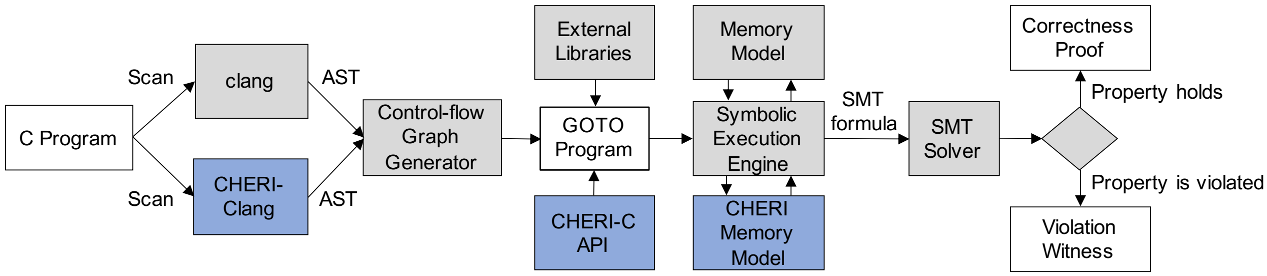

Figure 2: ESBMC architecture. The blocks with different background colours represent the standard ESBMC modules (grey), the CHERI-C extensions to ESBMC (blue) and the input/output blocks (white).

3) implementing the operational model for the CHERI API functions. These are described in the following sections.

Extension feature	Progress
Integration of CHERI-Clang front-end	75%
New types (`[u]intcap_t` and `*__capability`)	75%
CHERI-C API	50%
Cross-platform verification	100%
Pointer/integer casts	75%
Tagged memory support	0%
Internal representation of unions	100%
Bit-precise reasoning	100%

Table 1: List of necessary extensions (with their implementation status) to ESBMC for enabling hybrid CHERI-C support.

3.1 CHERI memory model in ESBMC

Implementing support for the CHERI memory model in ESBMC encompasses the following three steps: a) adding optional capability annotations to ESBMC's internal pointer representation, splitting addresses into those that represent capabilities and those that do not, b) introducing a new symbol representing the tagged memory containing the CHERI validity tags for allocated objects (i.e., those that may have their address taken), and updating it depending on the type of underlying assignments in the given program, and c) encoding the bounds and permission checks by constraints as required by the CHERI-enabled target platform (Morello, RISC-V or MIPS).

The ESBMC memory model is comprised of a collection of typed symbols (including *invalid* and *NULL* symbols). The symbols are uniquely identified during each step of the program's symbolic execution. Static and stack variables are represented as symbols of the corresponding type. At the same time, the dynamically allocated objects (e.g., via *malloc*) are stored as pointers to a symbolic array of bytes. ESBMC also uses these arrays to track allocations sizes and whether the dynamically allocated objects are still valid, or the underlying memory has been free'd (for identifying dangling pointers and memory leaks).

Pointer dereferencing is performed in multiple stages. Firstly, ESBMC performs a value set analysis to identify all possible targets for the dereferenced pointer. Secondly, ESBMC introduces bounds and temporal validity constraints for each identified target. Finally, ESBMC encodes the symbolic addresses of the potentially addressed objects and the corresponding constraints into a logical formula.

We extend ESBMC's memory model with capabilities by modelling them as tuples (obj_id, offset, metadata, tag) where metadata contains the (optionally compressed) in-memory representation of the additional information stored in CHERI capabilities (Figure 1).

3.2 Modelling CHERI-C API in ESBMC

Operational model for CHERI-C API. The CHERI-C API consists of about 50 public functions and macros intended to examine, modify and obtain capabilities. In CHERI-BSD [6] many are implemented by intrinsics understood by the CHERI-Clang compiler. As CHERI-enabled ESBMC leverages the same frontend, the current approach to modelling these intrinsics consists of formulating in ESBMC's operational model the semantics formally specified in the SAIL language [1] for the instructions generated by Clang, including conditions for hardware exceptions. This has been done for 8 intrinsics and the main functionality for examining capabilities.
Modified union representation. The extensive use of unions to access the individual fields shown in Figure 1 in the operational model of the CHERI-C API required reworking their internal representation in ESBMC. Specifically, instead of symbolically reasoning about their in-memory layout via byte arrays, they now provide direct access to the symbolic representation of its constituting types. While byte arrays did allow byte-level manipulations of unions via, e.g., `memset` to be expressed more naturally, they also suffered from repeated conversion from/to the in-memory representation of larger and especially structured members of unions such as CHERI capabilities in the operational model.
Bit-precise reasoning. Enabling bit-precise reasoning is another requirement for ESBMC since the CHERI-C API operational model extensively uses bit fields (e.g. function `cheri_length_get` in Listing 1). In ESBMC any object created dynamically (e.g., via *malloc*) is converted into an array of *byte* symbols (the smallest memory unit within the ESBMC IR), which means that all *struct* and *union* members not aligned to a byte cannot be uniquely addressed by their offsets (e.g., offsets of 1 and 7 bits will address the same *byte* symbol). This is resolved by applying masking and bit-shifting to the bytes fully containing the corresponding bit field.
Pure-capability CHERI-C support. The hybrid CHERI-C mode allows a co-existence of capabilities and standard pointers in programs, while in pure-capability mode all pointers are transparently treated as capabilities with the set of permissions and bounds determined by the compiler. In our extension of ESBMC we can now

express and reason about operations involving either capabilities or plain pointers simultaneously. Compiler-generated metadata for the former (e.g., when taking addresses of static or stack variables) is modelled in ESBMC using functionality from the CHERI-C API. Thus, support of pure-capability mode we get 'for free' from full support of the hybrid mode just by internally translating pointers to capabilities throughout.

4 RELATED WORK

ESBMC-CHERI is the first tool for formally verifying C programs for CHERI capability platforms.

CHERI can be compared to work which aims to extend C with fat pointers to obtain a memory safe variant of C. Two notable works here are CCured [15], CheckedC [18] which transform C programs into 'safe' versions, which would allow verified C code to be lifted to verified memory-safe C code but with the additional runtime overhead required to make such checks in software. Another set of approaches are dynamic analysis tools that perform instrumentation at the IR level (e.g. SoftBoundCETS [14, 13] and AddressSanitizer [17]) or source code level (e.g. MoveC [3]) to track pointers' metadata (*e.g.*, base, bound) using shadow space inspired mechanisms (instead of fat pointers).

CHERI is a next generation of development of these ideas, providing hardware support for capabilities, which is essential for their adoption in practice. There is a large stack of formal development for CHERI capability-enabled architecture, starting with formalizing ISA [1] to formally extending semantics of C with capabilities which is done in the Cerberus project [9]. Our work is built on top of these developments. At the same time, our work has a different angle to formalizing CHERI-C as we are aiming at automatically verifying end-user programs and ensuring their correct functioning on capability hardware. We suggest that the extensions presented in this paper can be adapted for implementing CHERI-C support in any bounded model checker with similar structure to ESBMC.

5 DISCUSSION AND FUTURE WORK

In this work we presented ESBMC-CHERI – an extension to ESBMC for verifying C programs for the CHERI capability hardware. We outlined the required extensions to the standard ESBMC: the integration of the CHERI-Clang front-end into ESBMC, the extension of the ESBMC memory model with CHERI capabilities and the implementation of CHERI-C API.

The next steps along the proposed extension scheme (as shown in Table 1) for ESBMC are: 1) implementing tagged memory, 2) completing the implementation of CHERI-C API, 3) completing the pure-capability CHERI-C support.

As soon as the above steps are finished, we will evaluate ESBMC-CHERI via, firstly, benchmarking on a set of over 15k C programs (hand-crafted or extracted from the real software) taken from SV-COMP. These programs will be interpreted as pure-capability CHERI-C programs and may exhibit different behaviour from standard C. Secondly, we will leverage the aforementioned DSbD software ecosystem to produce a set of industrially-relevant case studies. This evaluation of ESBMC-CHERI will be aimed at: 1) determining the scope of C programs that can be handled by ESBMC-CHERI, 2) identifying safe C programs that become unsafe or crash when executed on the CHERI-extended platform, and 3) determining how ESBMC-CHERI can be used to help the software developers adapting or writing new software to CHERI-enabled platforms.

ACKNOWLEDGMENT

The work in this paper is partially funded by the EPSRC grant EP/V000497/1. The data supporting this work are openly available from the GitHub repository at https://github.com/esbmc/esbmc/tree/cheri-clang.

REFERENCES

[1] Alasdair Armstrong et al. 2019. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. Proc. ACM Program. Lang. 3, POPL, Article 71.

[2] Armin Biere. 2009. Bounded model checking. In *Handbook of Satisfiability*, 457–481.

[3] Zhe Chen et al. 2019. Detecting memory errors at runtime with source-level instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 341–351.

[4] Edmund Clarke et al. 2004. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176.

[5] Lucas C. Cordeiro et al. 2012. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38, 4, 957–974.

[6] Brooks Davis et al. 2019. CheriABI: enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. Iris Bahar et al., editors. ACM, 379–393.

[7] Mikhail Y. R. Gadelha et al. 2018. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering ASE*. ACM, 888–891.

[8] Chris Lattner et al. 2004. LLVM: a compilation framework for lifelong program analysis and transformation. In *CGO*. San Jose, CA, USA, 75–88.

[9] Kayvan Memarian et al. 2019. Exploring C semantics and pointer provenance. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. Proc. ACM Program. Lang. 3, POPL, Article 67.

[10] Matt Miller. 2019. Trends and challenges in the vulnerability mitigation landscape. In USENIX Association, Santa Clara, CA.

[11] Felipe R. Monteiro et al. 2017. Bounded model checking of C++ programs based on the qt cross-platform framework. *Softw. Test. Verification Reliab.*, 27, 3.

[12] Felipe R. Monteiro et al. 2022. Model checking C++ programs. *Softw. Test. Verification Reliab.*, 32, 1.

[13] Santosh Nagarakatte et al. 2010. Cets: compiler enforced temporal safety for c. *SIGPLAN Not.*, 45, 8, 31–40.

[14] Santosh Nagarakatte et al. 2009. Softbound: highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 44, 6, 245–258.

[15] George C Necula et al. 2002. Ccured: type-safe retrofitting of legacy code. In *In 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 128–139.

[16] Phillipe A. Pereira et al. 2016. Verifying CUDA programs using SMT-based context-bounded model checking. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. Sascha Ossowski, editor. ACM, 1648–1653.

[17] Konstantin Serebryany et al. 2012. Addresssanitizer: a fast address sanity checker. In *2012 USENIX Annual Tech. Conf. (ATC'12)*, 309–318.

[18] David Tarditi et al. 2018. Checked c: making c safe by extension. In *IEEE Cybersecurity Development Conference 2018 (SecDev)*. IEEE, 53–60.

[19] Robert N. M. Watson et al. 2021. Assessing the Viability of an Open Source CHERI Desktop Software Ecosystem. Technical report. Capabilities Limited.

[20] Robert N.M. Watson et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, 20–37.

[21] Nathaniel Wesley Filardo et al. 2020. Cornucopia: temporal safety for CHERI heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, 608–625.

[22] Jonathan Woodruff et al. 2019. CHERI concentrate: practical compressed capabilities. *IEEE Transactions on Computers*, 68, 10, 1455–1469.

[23] Jonathan Woodruff et al. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st Int. Symposium on Computer Architecture (ISCA)*, 457–468.