IEEE SecDev | 2022

IEEE COMPUTER SOCIETY

Position Paper: Towards a Hybrid Approach to Protect Against Memory Safety Vulnerabilities

Kaled Alshmrany*, Ahmed Bhayat*, **Franz Brauße***, Lucas Cordeiro*, Konstantin Korovin*, Tom Melham^, Mustafa A. Mustafa*, Pierre Olivier*, Giles Reger*, Fedor Shmarov*

**The University of Manchester*

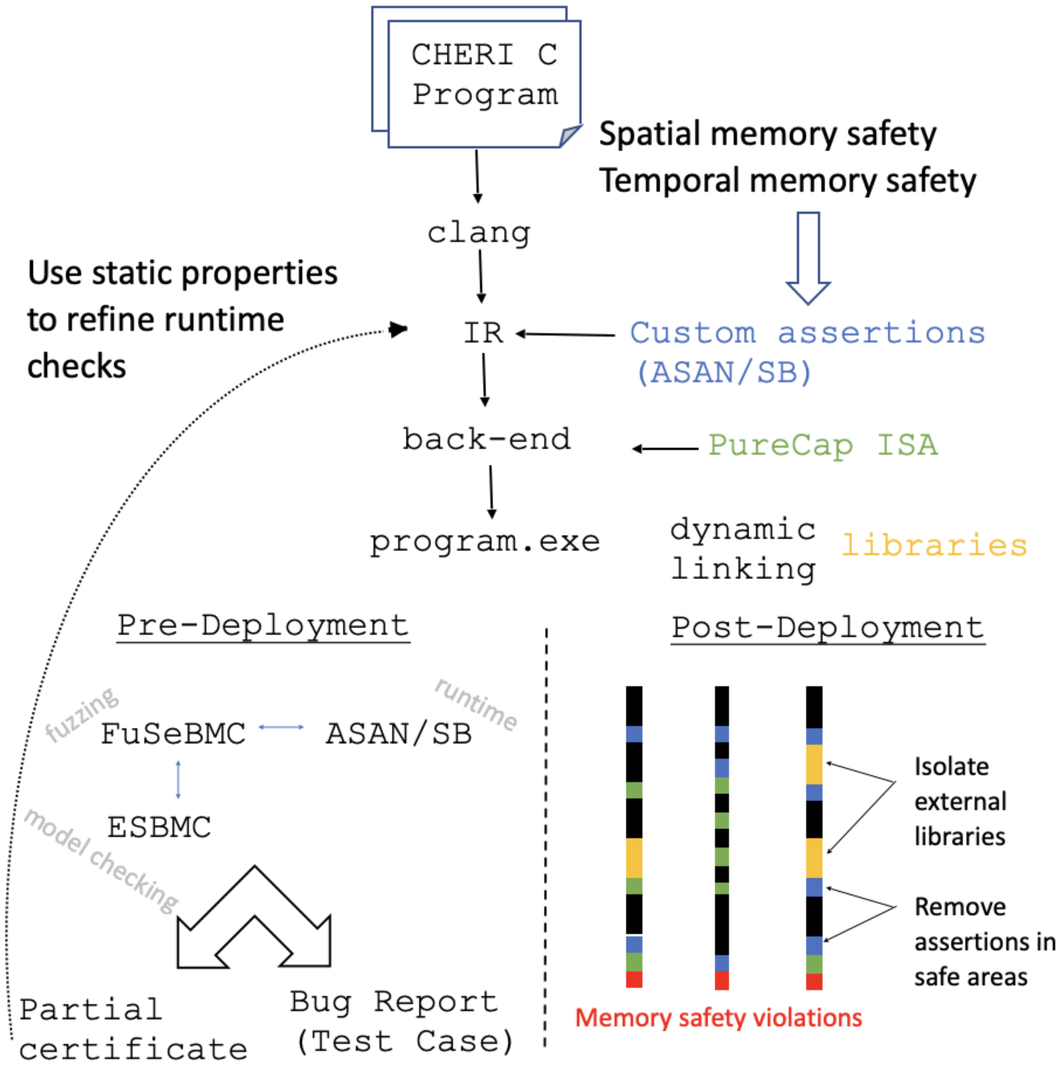
^University of Oxford

#IEEESecDev https://secdev.ieee.org/2022

Introduction

- There are many techniques for ensuring software safety
  - Based on static analysis, automated testing, etc.
- They provide different levels of protection
  - Some techniques can detect vulnerabilities that other cannot (and vise versa)
- Existing hybrid solutions (i.e., combining different techniques) showed promising improvements
  - Post-deployment software safety is often overlooked
- We propose a hybrid framework that addresses software safety across the pre-deployment and post-deployment stages

Hybrid Verification Framework Vision



- Accentuate post-deployment safety
  - Reduce performance overheads through "cheaper" hardware level protection
  - Reuse the information from static analysis to introduce only necessary "expensive" safety checks
- Enhance pre-deployment analysis
  - Combine complementary techniques
  - Avoid producing a monolithic hybrid solution (e.g., concolic execution)

Why hybrid and why now

- Why hybrid:
  - Different tools have different strengths and weaknesses
  - Existing solutions demonstrate very promising results
    - Frama-C, concolic execution, cooperative verification
    - However, all of them are for pre-deployment
  - We are addressing post-deployment **safety/performance balance**
- Why now:
  - Hardware memory protection techniques (e.g., CHERI) are being developed to cope with post-deployment performance overheads
    - But they provide a subset of safety guarantees
  - There are a lot of static analysis techniques that can be used for establishing partial safety of the program
    - This allows introducing targeted software hardening, thus increasing safety guarantees while keeping performance overheads manageable

Runtime Protection Techniques

- Can be used at pre-deployment + post-deployment
- Works on software and/or hardware level
- The resulting executable crashes on the inputs leading to the bug
- In software: based on program instrumentation (e.g., *AddressSanitizer*)
  - \+ Flexible for introducing new checks (properties)
  - \- Usually significant performance overheads
  - \- Functional equivalence of the instrumented program may be compromised
- In hardware: extended ISA + specialized compiler (e.g., *CHERI*)
  - \- Harder to introduce new checks
  - \+ Minor performance overheads
  - \- May introduce new semantics to the programs
- Needs concrete inputs for automated testing at pre-deployment
  - Usually combined with a sampling-based technique (e.g., fuzzing)

Static Analysis Techniques

- Used at pre-deployment only (in fact, ahead of compilation)
- Works with a mathematical abstraction of the underlying program
  - May lead to false-positives
- Analyzes the program with respect to the given specification and language semantics
- Requires assumptions (aka computational models) about the underlying hardware and system libraries
- The verification problem may be exponentially large or undecidable in general
- Often treated as a black-box: the program has a bug or is safe up to a point
  - The usefulness of the latter is often overlooked
  - We can reuse partial safety outcome (i.e., we can only say that a program is safe up to some execution depth)

Current Progress (analysis)

- We analyzed several runtime and static techniques on a subset of SV-COMP benchmarks (~ 300 C programs)
- Runtime Protection
  - AddressSanitizer – industry leader for detecting memory safety violations at runtime
  - SoftBoundCETS – tracks pointer bounds and temporal validity
  - CHERI PureCap – introduces pointer checks at hardware level
- Static Analysis
  - ESBMC – bounded model checker for single- and multi-threaded C/C++ programs
- Hybrid Techniques
  - FuSeBMC – combines BMC and fuzzing for automated test generation

Strengths and weaknesses

- Fuzzing helps BMC in bug-finding
- BMC is less scalable than runtime techniques
- BMC treats program's input symbolically, and it may prove program's safety
- Different runtime techniques find different types of vulnerabilities

Technique	Correct	Incorrect (FN+FP)	Timeout
ESBMC	107	3 (3+0)	17
FuSeBMC	116	2 (2+0)	9
Combined	116	2	9

Table 1. Programs requiring user inputs

Technique	Correct	Incorrect (FN+FP)	Timeout
ASAN	159	13 (13+0)	6
SB	152	20 (19+1)	6
PureCap	145	24 (24+0)	9
ASAN + SB	166	6 (5+1)	6
Runtime (combined)	166	6 (5+1)	6
ESBMC	130	5 (1+4)	43
FuSeBMC	133	4 (1+3)	41
Static (combined)	132	5 (1+4)	41

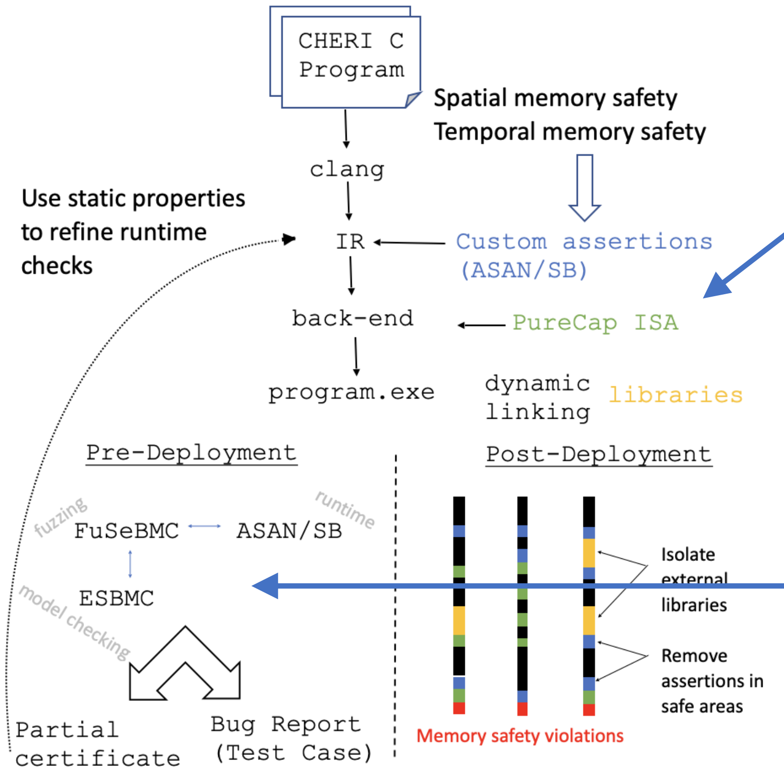
Table 2. Programs that do NOT require user-inputs

Different techniques find different bugs

Feature	ASAN	SB	PureCap (RISC-V)	ESBMC	FuSeBMC
Spatial Memory Safety					
Subobject buffer overflow	**no**	**no**	**no**/yes	yes	yes
Temporal Memory Safety					
Use-after-free	yes	yes	**no**	yes	yes
Stack use after return	**no**/yes	yes	**no**	yes	yes
Stack use after scope	yes	**no**	**no**	yes	yes
Double free	yes	yes	**no**	yes	yes
Memory leaks	yes	**no**	**no**	yes	yes
Program Features					
Unions	yes	yes	yes/**no**	yes	yes
Library functions	yes/no	yes/no	yes/no	yes/no	yes/no

Table 3. Qualitative analysis

Current progress (implementation)



ESBMC-CHERI [1] extends ESBMC to be able to reason about C/C++ programs that run on CHERI platforms.

- Modelling and handling inside ESBMC extra semantics that CHERI capabilities introduce

Exploring cooperation between BMC and fuzzing

- FuSeBMC [2] combining BMC and fuzzing for automated test generation
- EBF [3] combining BMC and fuzzing to verify concurrent C/C++ programs

[1] F. Brauße, F. Shmarov, R. Menezes, M. R. Gadelha, K. Korovin, G. Reger, and L. C. Cordeiro, "ESBMC-CHERI: towards verification of C programs for CHERI platforms with ESBMC," in *ISSTA 2022,* pp. 773–776.
[2] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "Fusebmc: An energy-efficient test generator for finding security vulnerabilities in c programs," in *TAP 2021,* p. 85–105.
[3] F. K. Aljaafari, R. Menezes, E. Manino, F. Shmarov, M. A. Mustafa, and L. C. Cordeiro, "Combining BMC and Fuzzing Techniques for Finding Software Vulnerabilities in Concurrent Programs", under review in *IEEE Access*

Future Work

- Using the information gathered during static analysis (i.e., partial safety outcome)
- Isolation of external libraries via hardware compartmentalization
- Exploring new verification techniques for incorporating into our hybrid framework
- Addressing functional equivalence of the program instrumentation for post-deployment

Thank you !!!