Verifying components of Arm® Confidential Computing Architecture with ESBMC

Tong Wu[1][0000−0002−0986−4150], Shale Xiong[2][0000−0001−9312−195X], Edoardo Manino[1][0000−0001−9312−195X], Gareth Stockwell[2][0009−0004−1773−2846], and Lucas C. Cordeiro[1,3][0000−0002−6235−4272]

[1] The University of Manchester, Manchester, UK
tong.wu-11@postgrad.manchester.ac.uk
{edoardo.manino,lucas.cordeiro}@manchester.ac.uk
[2] Arm®
{shale.xiong,gareth.stockwell}@arm.com
[3] Federal University of Amazonas, Manaus, Brazil

Abstract. *Realm Management Monitor* (RMM) is an essential firmware component within the recent Arm *Confidential Computing Architecture* (Arm CCA). Previous work applies formal techniques to verify the specification and prototype reference implementation of RMM. However, relying solely on a single verification tool may lead to the oversight of certain bugs or vulnerabilities. This paper discusses the application of ESBMC, a state-of-the-art Satisfiability Modulo Theories (SMT)-based software model checker to further enhance RRM verification. We demonstrate ESBMC's ability to precisely parse the source code and identify specification failures within a reasonable time frame. Moreover, we propose potential improvements for ESBMC to enhance its efficiency for industry engineers. This work contributes to exploring the capabilities of formal verification techniques in real-world scenarios and suggests avenues for further improvements to better meet industrial verification needs.

Keywords: Formal verification · Software model checking · Software testing · Firmware · Security.

1 Introduction

The rise of Confidential Computing is driven by the need to secure computations in Cloud Computing, where sensitive information is delegated to a third party, posing potential security risks [22]. For example, hundreds of millions of Facebook user records were exposed on Amazon cloud server [20].

Confidential Computing technologies, including Arm *Confidential Computing Architecture* (Arm CCA) [15], introduce protected execution environments, for example *Realms* in Arm CCA, to ensure confidentiality and integrity of sensitive information. In particular, Arm CCA allows multiple Realms to execute in parallel; to manage all instances, Arm CCA then introduces a new privileged firmware component called *Realm Management Monitor* (RMM). RMM also

provides a necessary interface to the outside, non-secure world. Arm is committed to specifying the behavior of RMM [16] and providing an implementation in C, and the code is already open-sourced to the community [17].

RMM is a critical component, and any end-user must trust it. Arm provides a specification [16] for all the interfaces via pairs of pre/post conditions. To verify the correctness of the RMM implementation concerning the RMM specification, formal methods such as interactive theorem provers and symbolic model checking can be applied to automate the verification workflow [14]. Arm engineers automatically generate a verification harness from a machine-readable specification of the RMM by evaluating the pre- and post-conditions to constrain the inputs and outputs. This harness can be consumed by a software verification tool such as *C Bounded Model Checker* (CBMC) [11, 13]. Arm researchers and engineers are gradually deploying these auto-generated verification harnesses and CBMC in the CI/CD system. Several violations in the implementations detected by CBMC have been confirmed and fixed by Arm engineers, thus demonstrating the value of software model checking techniques [7] in strengthening safety-critical systems.

Given the fact that there exist some violated properties in the current draft implementation of the RMM [7], we are keen to explore the following questions:

- Is the existing verification enough to secure RMM?
- If not, can other state-of-the-art techniques find additional violations?

To explore these questions, we further apply ESBMC, an efficient model checker based on SMT theories that can automatically detect or prove the absence of runtime errors in software written in C/C++, Kotlin, Python, and Solidity [9, 18]. According to the recent Competitions on Software Verification (SV-COMP) [2, 3], ESBMC consistently outperforms CBMC in proving more safety properties while producing fewer incorrect results. However, there is little evidence of whether this superior performance could uncover more vulnerabilities in real-world low-level software systems. Through our exploration of ESBMC on some RMM verification cases, we achieved the following contribution:

- We reproduced the same failures reported by CBMC, which were confirmed by Arm engineers previously.
- We identified inconsistent results from CBMC and reported them to the developers.
- We found 23 new violations in the RMM code that only ESBMC can detect.
- We showed that the verification performance could be significantly improved by efficiently configuring the bounds for each loop in the program.
- We highlighted the challenge of checking multiple properties and contributed to its development within ESBMC.

2 Background

2.1 Bounded Model Checking (BMC)

SAT/SMT-based BMC is a formal verification technique to falsify or prove the correctness of finite-state systems [5]. This method explores a system's state

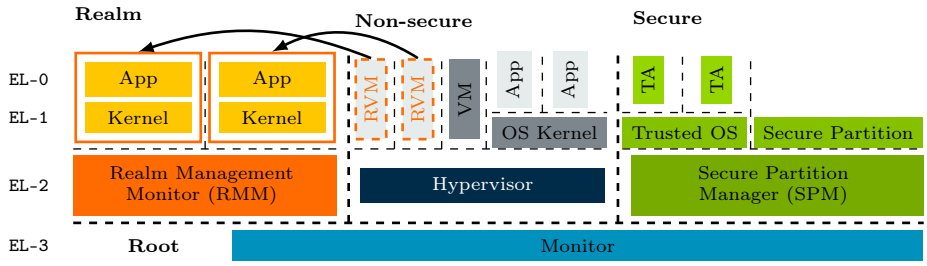

Fig. 1. An architecture of an Arm CCA [7]. The physical memory space is separated into four worlds: Non-secure, Secure, Root and the Realm.

space up to a specified depth or bound k, searching for potential errors or violations within that limited scope. Then the program is encoded into an SAT/SMT formula, and it is satisfiable *iff* there exists a counterexample that violates the property [4].

As a BMC tool, ESBMC implements various algorithms for bounded and incremental verification [1, 6, 8]: (1) incremental BMC, which unrolls the loops and checks the property incrementally; (2) k-induction [10] proof-rule algorithm, which aims to prove the safety based on induction; and (3) default BMC approach that bounds the program up to a given depth and call the SMT solver once to solve the entire SMT formula.

Similar to CBMC, ESBMC takes an RMM test case as input and checks the safety of the properties in the given program. In particular, ESBMC can check for user-defined assertions, pointer safety, buffer overflow, data races, and so on through configurations. The ESBMC documentation about the supported safety/security properties, underlying verification algorithms, and integrated SMT solvers is available online.[4]

2.2 Realm Management Monitor

Fig. 1 illustrates the architecture of Arm CCA; more can be learned from [15]. Executions Arm's *Processing Element* (PE) is associated with levels EL0 to EL3, where user-space executes at EL0 and most privileged low-level firmware, or monitor, executes at EL3. The physical memory space also separates into four parts, or four worlds, namely, *Non-secure* World, *Secure* World, *Root* World, and the newly-introduced *Realm*. In the *Realm* World, an end-user application, referred to as a realm, is executed at the level of EL0 and EL1. To administer Realms, Arm CCA introduces a new privileged firmware component called the *Realm Management Monitor* (RMM), executing at EL2 in the Realm world, which acts as a separation kernel isolating Realms from each other. Also, RMM

[4] `https://ssvlab.github.io/esbmc/documentation.html`

D3.2.5 RMI_GRANULE_DELEGATE

Delegates a Granule.

D3.2.5.1 Interface

D3.2.5.1.2 Input Values

Name	Register	Field	Type	Description
fid	X0	[63:0]	UInt64	Command FID
addr	X1	[63:0]	Address	PA of the target Granule

D3.2.5.1.3 Output Values

Name	Register	Field	Type	Description
result	X0	[63:0]	ReturnCode	Command return status

D3.2.5.2 Failure conditions

ID	Condition
gran_align	pre: !AddrIsGranuleAligned(addr)
	post: ResultEqual(result,RMI_ERROR_INPUT)

(–– continued in the right column)

(–– from the left column)

gran_bound	pre: !PaIsDelegable(addr)
	post: ResultEqual(result,RMI_ERROR_INPUT)
gran_state	pre: Granule(addr).state != UNDELEGATED
	post: ResultEqual(result,RMI_ERROR_INPUT)
gran_pas	pre: Granule(addr).pas != NS
	post: ResultEqual(result,RMI_ERROR_INPUT)

D3.2.5.3 Success conditions

ID	Post-condition
gran_state	Granule(addr).state == DELEGATED
gran_pas	Granule(addr).pas == REALM

D3.2.5.4 Footprint

ID	Value
gran_state	Granule(addr).state
gran_pas	Granule(addr).pas

Fig. 2. RMI_GRANULE_DELEGATE command specification (draft) [7]

provides interfaces to the Non-secure World for managing and scheduling realms indirectly.

Fig. 2 illustrates an example of RMM specification of the RMI_GRANULE_DELEGATE command.[5] The specification takes a single granule address as input from the Host (the *addr* parameter, in general purpose register *X1*, is the physical address location of the granule to be delegated). If the corresponding granule metadata is UNDELEGATED, meaning that the granule is not in the REALM PAS, and it is currently in the NS PAS, then the granule is moved to the REALM PAS. The failure conditions consist of several pairs of pre/post conditions, each of which is a possible error condition. When no failure occurs, the success conditions describe the expected updates to the RMM machine state.

3 RMM Verification with ESBMC

Listing 1.1 illustrates a C example of a verification harness generated from the RMM specification. It initializes a non-deterministic global state from line 1 to 7. In line 9, it executes the actual command of the specification. Line 8 and line 10 evaluate the pre- and post-conditions using the Boolean variables *failure_src_align_pre* and *failure_src_align_post*, respectively. Lines 11-16 are the boolean conditions of the pre- and post-conditions. Finally, the assertion in line 17 checks the post-condition if the pre-condition fails. From this verification harness, we can use any off-the-shelf software model checker to automatically check the assertion for the post-condition. Properly choosing the underlying verification strategy and parameters to configure ESBMC is essential to achieving efficient verification of RMM. Here, we introduce two main verification techniques available in ESBMC: *bounded verification* to unwind each loop occurring

[5] This is an example from the drafted specification version accessed in early 2022 [17].

in the program with a different upper bound and *multi-property check* to verify all properties incrementally using the underlying SMT solver.

```
1  struct tb_regs __tb_regs = __tb_arb_regs();
2  __tb_regs.X0 = SMC_RMM_DATA_CREATE;
3  __tb_regs.X1 = nondet_uint64_t(); // data
4  __tb_regs.X2 = nondet_uint64_t(); // rd
5  __tb_regs.X3 = nondet_uint64_t(); // map_addr
6  __tb_regs.X4 = nondet_uint64_t(); // src
7  __init_global_state(__tb_regs.X0); // Generate non-deterministic state
8  bool failure_src_align_pre = !AddrIsGranuleAligned(src); // Precondition
9  uint64_t result = tb_handle_smc(&__tb_regs); // Execute command
10 bool failure_src_align_post = ResultEqual(result, RMI_ERROR_INPUT); //
       Postcondition
11 // Failure condition assertions (excerpt)
12 bool prop_failure_src_align_ante = failure_src_align_pre;
13 __COVER(prop_failure_src_align_ante);
14 if (prop_failure_src_align_ante) {
15 bool prop_failure_src_align_cons = failure_src_align_post;
16 __COVER(prop_failure_src_align_cons);
17 __ASSERT(prop_failure_src_align_cons , "prop_failure_src_align_cons"); }
```

Listing 1.1. Verification harness from the specification [7]

3.1 Bounded Verification

We can configure the loops' unrolling depth for BMC tools to reduce the program state space for refutation/verification. In particular, we must limit the unwinding bound to avoid an unbounded loop being unrolled infinitely. However, we must carefully configure the bounds for complex programs such as with nested loops because the verification time could increase dramatically. Both ESBMC and CBMC support setting different upper bounds for each loop in the program. We rely on the manual annotations of ARM engineers, who can use their extensive knowledge of the RMM code to set different bounds and unwind assertions for each loop. While this setup can be complicated and time-consuming compared to setting a unified bound for all loops or unrolling them incrementally, it can result in smaller state spaces and faster checking speeds, as shown in our experimental evaluation. This happens because iteratively unrolling loops and calling a solver via incremental and induction methods can be costly, especially for programs containing various nested loops. We refer the reader to our previous work about handling loops in BMC for a detailed discussion about this topic [1, 8], including comparisons to state-of-the-art verification methods. To verify the components of the Arm Confidential Computing Architecture, which is the focus of this paper, our preliminary experiments on incremental BMC have shown a significant performance difference when we set different upper bounds for each loop, which will be presented in Section 4.

3.2 Multi-Property Check

Most real-world verification instances contain multiple assertions [21]. This is because any non-trivial program must satisfy several invariants at once. However, multiple assertions create a crucial dilemma for software verification tools:

```
#include <assert.h>
extern int nondet_int();
int main() {
  int a = nondet_int();
  switch (a) {
  case 0: assert(a > 0); break;
  case 1: assert(a > 1); break;
  default: return 0;
  }
}
```

Listing 1.2. A program with two property violations.

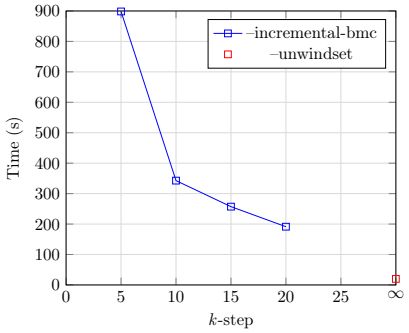

Fig. 3. Verification time for different k-steps during each loop rolling. The **red** node at the bottom right represents unrolling the loops completely until the given bounds.

whether we encode them all in a single large formula or split them into separate smaller ones. A single large formula is faster to solve than smaller ones because the latter have multiple (or incremental) queries to a solver [18]. However, it also raises the problem that the counterexample in a single formula can only expose one property violation in the program.

By default, ESBMC runs in single property check mode, which encodes all verification conditions (VCs) into one single SMT formula and terminates once if it finds a violation by the underlying SMT solver across the program path. ESBMC uses Boolector as its default SMT solver [19]. Recently, ESBMC added verification support for multi-property checks (see Listing 1.2) to report all property violations for a single call. During the multi-property check, each property assumes the other properties are unreachable by keeping the current encoded property, i.e., all the other assertions are removed as if they do not exist. In the example of Listing 1.2, none of the two assertions hold because the value of a is non-deterministic and can reach both cases to trigger the assertion. ESBMC, in single property check mode, will terminate once it finds a violation, e.g., `assert(a>1)`, while multi-property check mode aims at reporting both of these violations.

4 Experiment Results

In this section, we evaluate the capability of ESBMC single- and multi-property checks on RMM.

Experimental Setup. We conduct the experiments on a Ubuntu OS with 16GB memory and an Intel i7 processor, limiting the timeout to 5000 s. The benchmarks are listed in Table 1 and are open-source [17]. We compare the results

between ESBMC v7.4 and CBMC v5.94. Unless we state otherwise, we run ESBMC with the flag `-unwindset`. This flag ensures it uses the same default approach of CBMC, i.e., unrolling each loop to the maximum number of loop iterations occurring in the program without producing under-approximation. ESBMC adds explicit assertions to check whether the loop is fully unrolled.

Availability of Data and Tools. All tools, benchmarks, and evaluation results are available on a supplementary page: `https://zenodo.org/records/13255044`.

Number of Violations Found. Table 1 shows the total number of failures found by ESBMC and CBMC. The main finding is that ESBMC reports more failures than CBMC in three test cases.

Confirmed and Unconfirmed Violations. We are still working on confirming every bug reported by ESBMC due to the complexity and readability of the output traces. In particular, ESBMC provides traces that describe hundreds or thousands of states, where each state contains information about the program location and the value of the local/global variables. As Table 1 shows, ESBMC finds additional bugs in the implementation of three primitives: `RMI_REALM_ACTIVATE`, `RMI_REALM_DESTROY` and `RMI_DATA_DESTROY`. We are in constant contact with the developers of RMM to track their progress, and currently ARM developers have already confirmed the presence of bugs in `RMI_REALM_DESTROY`, which is caused by converting a pointer to an integer and can affect the global verification conditions in that test case. The actual RMM code was fixed, though waiting for code review, before we conducted this experiment, as it involved undefined behavior due to that conversion. We present a characterized example in Listing 1.3, and we have reported the issue to the CBMC team so that they could improve their verifier. A CBMC developer has confirmed the bug in their memory model (see `https://github.com/diffblue/cbmc/issues/8161`).

Impact of step k in incremental BMC. ESBMC provides an option to set the k-step when incrementally unrolling the loops. For example, a k-step of 5 will unfold a given loop incrementally as 5, 10, 15, and 20 (maximum k-step). We conduct an extra experiment on a single but representative RMM test case to check the impact of step k in incremental BMC. In Fig. 3, it can be seen that although increasing the k-steps can reduce looping unrolling and solving time since we can find a property violation faster, it is not as efficient as ESBMC non-iterative mode, i.e., via unwindset that unrolls the loops directly to the upper bound. These results explain our chosen methodology as described in Section 3.1.

```
#include<assert.h>
int arr[8] = {1,2,3,4,5,6,7,8};
int main() {
  int *a = &arr[7];
  if((unsigned long)a >= (unsigned long)(-4095))
    assert((unsigned long)(-1*(long)a) < 6); //esbmc fails, cbmc  success
}
```

Listing 1.3. Pointer to integer convertion example extracted from rmm, where the assertion should fail if the address falls into a negative value, for example, -2048.

Table 1. Verification results of multi-property checks using ESBMC and CBMC.

Command	Assert Fail		VCCs/Solver Calls	
	ESBMC	CBMC	ESBMC	CBMC
RMI_REC_DESTROY	20	20	113/113	142/19
RMI_GRANULE_DELEGATE	safe	safe	54/54	132/2
RMI_GRANULE_UNDELEGATE	1	1	45/45	132/1
RMI_REALM_ACTIVATE	**3**	**safe**	53/53	140/1
RMI_REALM_DESTROY	**17**	**1**	114/114	148/2
RMI_REC_AUX_COUNT	1	1	48/48	139/2
RMI_FEATURES	safe	safe	21/21	125/1
RMI_DATA_DESTROY	>=**26**	**22**	82/82	151/18

Performance Comparison. Fig. 4 illustrates the verification time of the test cases for three runs: ESBMC single property check, CBMC multi-property check, and ESBMC multi-property check. Note that the default option of CBMC always checks all properties in one execution, which is relevant when verifying industrial code. CBMC and ESBMC single property checks can finish most verification tasks within a few seconds. However, the ESBMC multi-property check needs a few minutes for each task and even longer for the last case in the figure, which has exceeded the timeout of 5000 s. Table 1 provides an insight into the performance difference. ESBMC naively calls an SMT solver when there is a generated SMT formula (per property to check), and it takes time to solve them individually. Meanwhile, CBMC can exploit the power of incremental SAT solving by MiniSAT to reuse solved instances for new problems [12]. Another important factor is that CBMC implements a string solver to solve frequent string operations at RMM code, while ESBMC has not implemented one yet, which results in a performance slowdown for strings.

```
...
case SMC_RMM_RTT_READ_ENTRY:
    struct smc_result rst;
    smc_rtt_read_entry(*X1, *X2, *X3, &rst);
    result = rst.x[0]; *X1 = rst.x[1]; *X2 = rst.x[2];
    *X3 = rst.x[3]; *X4 = rst.x[4];
    break;
...
```

Listing 1.4. Code segement did not compile for Clang since it cannot declare variables inside a case block without enclosing it in braces.

Syntax Errors. In addition to the correctness and performance, there exists an interesting aspect related to the deployed front-end parsers: CBMC uses a modified C parser, while ESBMC implements Clang API to transform the source code into Clang AST [9], without having details of the input program compiled away. The latter can do strict and elegant code syntax checks before starting model checking. ESBMC can also provide compilation error messages as expected from a compiler and leverage Clang's static analyzer to provide meaningful warnings

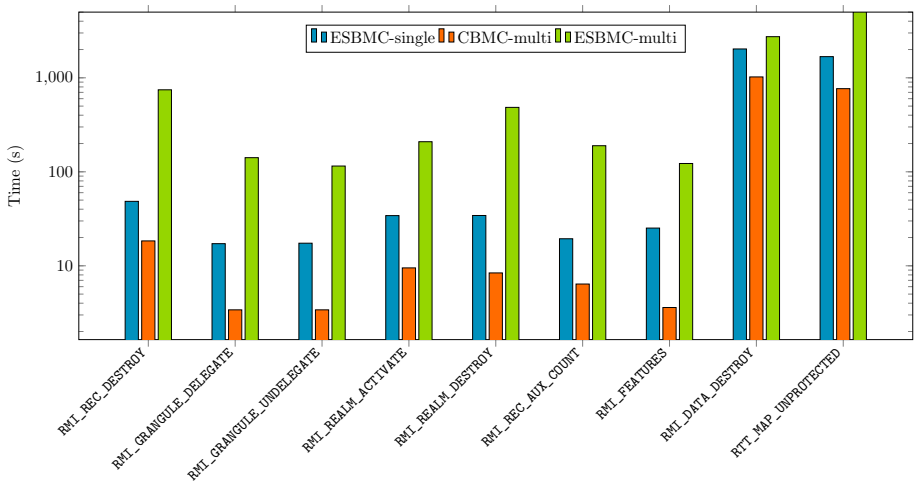

Fig. 4. Experiment result, where x-axis represents the RMM testcases, and y-axis represents the verification time in seconds. Different colors: blue on the left is for ESBMC single-property check, red on the middle is for CBMC multi-property check, and yellow on the right is for ESBMC multi-property check.

when parsing the program. For example, consider Listing 1.4 that illustrates an example from the RMM implementation, where a declaration of **struct** object is in the switch case but without curly braces. ESBMC reports the error using Clang, while the C parser in CBMC ignores it.

5 Conclusions

We present our application of ESBMC to verify newly introduced RMM components in Arm CCA. First, we show that to ensure firmware correctness, we can fully verify it by applying various techniques, each with its strengths and weaknesses. In our experiment, we applied ESBMC and found more violation properties than CBMC. Second, we show the community that we must write more scalable verifiers to check large and complex code bases for multi-property checks. Third, we show that relying on manual annotations for loop unrolling is still superior to automated techniques such as incremental bounded model checking. Fourth, we recommend that software verifiers connect their verification algorithms with industrial-strength compilers, as we find that self-developed parsers may not be precise enough. Lastly, we encourage tool developers to focus on producing more readable verification results which would improve the usability of formal verification techniques in the software industry. These insights inform our current plan for improving the scalability of ESBMC for verifying industrial code. More specifically, we plan to include faster algorithms for multiple property verification (e.g., transforming assertions into assumptions and

grouping properties) and automate the loop unwinding process with data-driven algorithms such as machine learning.

Acknowledgments

The work in this paper is partially funded by the Arm Center of Excellence at the University of Manchester, UK, EPSRC grants EP/T026995/1, EP/V000497/1, EP/X037290/1, EU H2020 ELEGANT 957286, and Soteria project awarded by the UK Research and Innovation for the Digital Security by Design (DSbD) Programme. We acknowledge Franz Brauße and Chenfeng Wei, who helped refine ESBMC for RMM verification. We would also like to thank engineers from `tf-rmm` team (`https://www.trustedfirmware.org/projects/tf-rmm/`).

A Appendix

This section lists the details from above results where ESBMC reports more failures from RMM testcases than CBMC.

Table 2. Unique results from ESBMC

No.	Location	Description
1	`tb_rmi_realm_activate.c` line 107	`prop_success_realm_state_cons`
1	`tb_rmi_realm_activate.c` line 98	`prop_result_cons`
1	`tb_rmi_realm_activate.c` line 89	`prop_failure_realm_state_cons`
2	`tb_rmi_realm_destroy.c` line 123	`prop_success_rd_state_cons`
2	`tb_rmi_realm_destroy.c` line 115	`prop_success_rtt_state_cons`
2	`tb_rmi_realm_destroy.c` line 106	`prop_result_cons`
2	`granule.h` line 70	assertion `false`
2	`granule.h` line 66	assertion `g->refcount == 0UL`
2	`granule.h` line 63	assertion `g->refcount <= GRANULE_SIZE / sizeof(uint64_t)`
2	`granule.h` line 59	assertion `g->refcount == 0UL`
2	`granule.h` line 56	assertion `granule_refcount_read_relaxed(g) <= 1UL`
2	`granule.h` line 46	assertion `g->refcount == 0UL`
2	`granule.h` line 43	assertion `granule_refcount_read_relaxed(g) == 0UL`
2	`tb_lock.c` line 80	The granule must be locked.
2	`granule.h` line 120	assertion locked
2	`tb_lock.c` line 49	The granule lock must be free.
2	`status.h` line 90	assertion `(unsigned int)(-1 * (long)ptr) < RMI_ERROR_COUNT`
2	`vmid.c` line 63	assertion `vmid < vmid_count`
2	`realm.c` line 337	unwinding assertion loop
3	`granule.h` line 120	assertion locked
3	`granule.c` line 47	assertion `idx < RMM_MAX_GRANULES`
3	`tb_lock.c` line 49	The granule lock must be free.
3	`s2tt.c` line 358	assertion `map_addr < (1UL « ipa_bits)`
3	`s2tt.c` line 357	assertion `level >= start_level`
3	`s2tt.c` line 356	assertion `start_level >= MIN_STARTING_LEVEL`

References

[1] Omar M. Alhawi et al. "Verification and refutation of C programs based on k-induction and invariant inference". In: *Int. J. Softw. Tools Technol. Transf.* 23.2 (2021), pp. 115–135.

[2] Dirk Beyer. "Competition on Software Verification and Witness Validation: SV-COMP 2023". In: *TACAS. LNCS*. Vol. 13994. Springer, 2023, pp. 495–522. DOI: 10.1007/978-3-031-30820-8_29. URL: https://doi.org/10.1007/978-3-031-30820-8%5C_29.

[3] Dirk Beyer. "State of the Art in Software Verification and Witness Validation: SV-COMP 2024". In: *30th International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14572. LNCS. Springer, 2024, pp. 299–329.

[4] Armin Biere. "Bounded Model Checking". In: *Handbook of Satisfiability*. Vol. 185. IOS Press, 2009, pp. 457–481. DOI: 10.3233/978-1-58603-929-5-457. URL: https://doi.org/10.3233/978-1-58603-929-5-457.

[5] Edmund M. Clarke et al. "Bounded Model Checking Using Satisfiability Solving". In: *Formal Methods Syst. Des.* 19.1 (2001), pp. 7–34. DOI: 10.1023/A:1011276507260. URL: https://doi.org/10.1023/A:1011276507260.

[6] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. "SMT-Based Bounded Model Checking for Embedded ANSI-C Software". In: *IEEE Trans. Software Eng.* 38.4 (2012), pp. 957–974.

[7] Anthony C. J. Fox et al. "A Verification Methodology for the Arm® Confidential Computing Architecture: From a Secure Specification to Safe Implementations". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 376–405. DOI: 10.1145/3586040. URL: https://doi.org/10.1145/3586040.

[8] Mikhail Y. R. Gadelha, Hussama Ibrahim Ismail, and Lucas C. Cordeiro. "Handling loops in bounded model checking of C programs via k-induction". In: *Int. J. Softw. Tools Technol. Transf.* 19.1 (2017), pp. 97–114. DOI: 10.1007/S10009-015-0407-9. URL: https://doi.org/10.1007/s10009-015-0407-9.

[9] Mikhail Y. R. Gadelha et al. "ESBMC 5.0: an industrial-strength C model checker". In: *ASE*. ACM, 2018, pp. 888–891.

[10] Mikhail Y. R. Gadelha et al. "ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference - (Competition Contribution)". In: *TACAS. LNCS*. Vol. 11429. Springer, 2019, pp. 209–213. DOI: 10.1007/978-3-030-17502-3_15. URL: https://doi.org/10.1007/978-3-030-17502-3%5C_15.

[11] Amazon Inc. *AWS CBMC Viewer*. https://github.com/model-checking/cbmc-viewer. Last viewed June 2022. 2022.

[12] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science.

An EATCS Series. Springer, 2016. DOI: 10.1007/978-3-662-50497-0. URL: https://doi.org/10.1007/978-3-662-50497-0.

[13] Daniel Kroening and Michael Tautschnig. "CBMC - C Bounded Model Checker - (Competition Contribution)". In: *TACAS, LNCS*. Vol. 8413. Springer, 2014, pp. 389–391. DOI: 10.1007/978-3-642-54862-8_26. URL: https://doi.org/10.1007/978-3-642-54862-8%5C_26.

[14] Xupeng Li et al. "Design and Verification of the Arm Confidential Compute Architecture". In: *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 2022, pp. 465–484. URL: https://www.usenix.org/conference/osdi22/presentation/li.

[15] Arm Ltd. *Arm Confidential Compute Architecture*. https://www.arm.com/architecture/security-features/armconfidential-compute-architecture. 2021.

[16] Arm Ltd. *Realm Management Monitor beta0 Specification*. https://developer.arm.com/documentation/den0137/a/. Accessed 25th October 2022, final version to be published 2022. 2022.

[17] Arm Ltd. *TF-RMM Homepage*. https://www.trustedfirmware.org/projects/tf-rmm/. Last accessed 14 Jan 2024.

[18] Rafael Sá Menezes et al. "ESBMC v7.4: Harnessing the Power of Intervals - (Competition Contribution)". In: *TACAS, LNCS*. Vol. 14572. Springer, 2024, pp. 376–380. DOI: 10.1007/978-3-031-57256-2_24. URL: https://doi.org/10.1007/978-3-031-57256-2%5C_24.

[19] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0". In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 53–58. DOI: 10.3233/SAT190101. URL: https://doi.org/10.3233/sat190101.

[20] Jason Silverstein. "Hundreds of millions of Facebook user records were exposed on Amazon cloud server". In: *CBS News* 4 (2019).

[21] Janislley Oliveira de Sousa et al. "Finding Software Vulnerabilities in Open-Source C Projects via Bounded Model Checking". In: *CoRR* abs/2311.05281 (2023). DOI: 10.48550/ARXIV.2311.05281. URL: https://doi.org/10.48550/arXiv.2311.05281.

[22] S. Subashini and V. Kavitha. "A survey on security issues in service delivery models of cloud computing". In: *J. Netw. Comput. Appl.* 34.1 (2011), pp. 1–11. DOI: 10.1016/J.JNCA.2010.07.006. URL: https://doi.org/10.1016/j.jnca.2010.07.006.