Towards Comprehensive Verification of Hardware and Software for RISC-V based Embedded Systems*

Niklas Bruns[1], Sallar Ahmadi-Pour[1], Sören Tempel[1], Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
nbruns@uni-bremen.de, sallar@uni-bremen.de, tempel@uni-bremen.de, drechsler@uni-bremen.de

Abstract

In this extended abstract, we present several approaches for the verification of embedded systems. First, we present four cross-level approaches for processor verification at the *Register-Transfer Level* (RTL) using a cross-level setting with an *Instruction Set Simulator* (ISS) as a reference model. Second, we present a comprehensive verification approach for verifying embedded software with symbolic execution.

1 Introduction

With the rising complexity of modern embedded systems, the verification gap becomes an essential factor in the design flow. As embedded systems deal with intricate interactions between software and hardware, the verification methods should consider these interactions as part of the system under verification. Modern design flows for such integrated systems can utilize *Virtual Prototypes* (VP), which allow early software development before the physical hardware exists. Early software development as well as the functional reference for both software and hardware engineers helps to reduce the gap between design and development stages. The integration of verification methods throughout the design and development stages allows the early mitigation of bugs. In this extended abstract, we present our recent verification approaches that help bridge the aforementioned verification gap and help verify the complex interaction between software and hardware of embedded systems.

In Fig. 1 we illustrate how our verification methods are integrated around a VP-based design flow. Our cross-level processor verification approaches aim to verify the RTL design under test, and use the ISS of the VP as functional reference model. The test generation techniques of the proposed verification approaches are based on random testing, coverage-guided fuzzing, and symbolic execution. More details on cross-level processor verification are provided in Section 2. Furthermore, we also leverage VPs for embedded software verification. To this end, we have implemented a custom ISS which symbolically executes the software under test based on symbolic inputs injected via peripherals provided by the VP. Symbolic execution allows us to maximize path coverage and check for error conditions on each executed path. Refer to Section 3 for more information.

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127.

2 Cross-Level Processor Verification

The modular and extensible open-source *Instruction Set Architecture* (ISA) RISC-V [1, 2], which is very popular in industry and academia, enables royalty-free processor design and implementation. However, this modularity and extensibility adds verification complexity as verification tools must handle the large configuration space and microarchitecture-specific optimizations. Recently, approaches tailored explicitly for RISC-V verification have emerged. The baselines are the official RISC-V unit and compliance tests [3, 4], which are directed test suites. An alternative approach is Google's open-source RISC-V *Design Verification* (DV) framework, which uses a co-simulation that employs an ISS as a functional reference model for the RTL processor under test and applies constraint-based specification techniques to generate individual RISC-V assembly test files. Execution results between the ISS and RTL processor core are evaluated through an execution log file comparison.

While RISC-V DV is very powerful in general, it still has some significant weaknesses. It only uses short instruction sequences, and the employed instruction set is restricted to bypass problems with infinite loops and platform-dependent memory access operations. Furthermore, it has an extensive filesystem communication performance overhead because each test file must be compiled, loaded, and each executed test also creates a log file for compar-

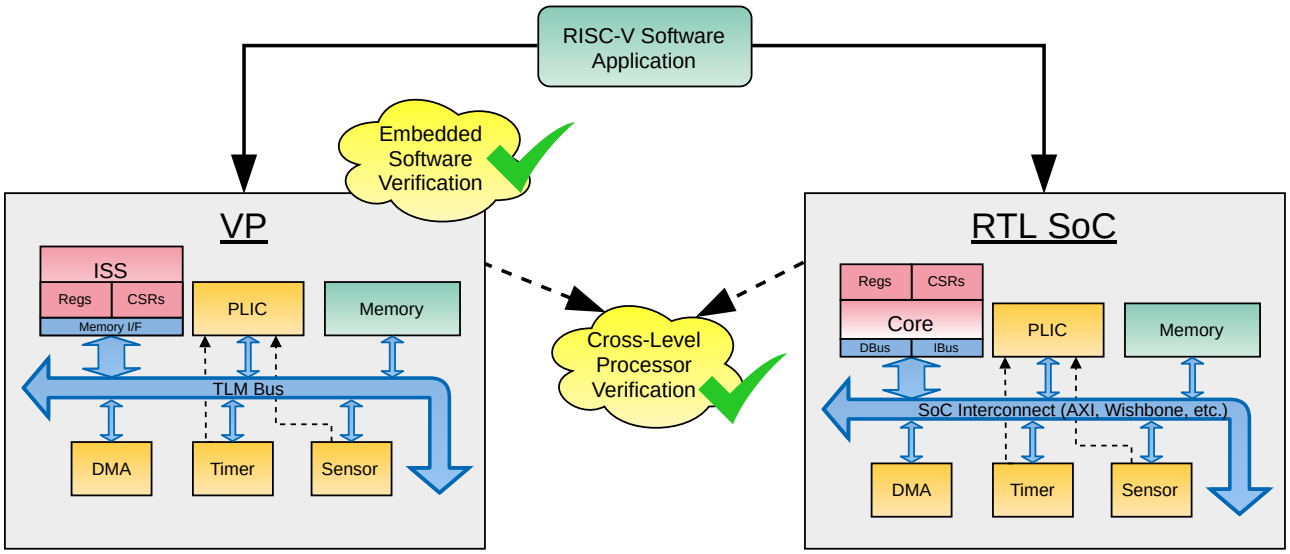

Figure 1 Conceptual overview of our hardware and software verification techniques for RISC-V embedded systems.

ison. Last but not least, the test generator is not guided by coverage. We investigated the quality of generated tests of RISC-V DV through a mutation-based comparison between a reference ISS and a mutated ISS in [5].

In [6], we addressed many of the abovementioned issues. Our approach generates endless instruction streams and tightly integrates the ISS with the RTL core with the aid of in-memory communication. The setup allows a restriction-free instruction generation, which enables a very comprehensive test approach.
However, the approach still does not use the runtime coverage to guide the test generation process. Instead, it is based on a simple randomized test strategy that makes it very difficult to continuously achieve a broad and deep test coverage in endless instruction streams. In [7], we proposed a novel cross-level verification approach that addresses this limitation by dynamically evolving the instruction stream at runtime based on observed coverage information. Furthermore, the novel concept of coverage-guided aging is employed to smooth out the coverage distribution over time. Our experiments with an industrial RISC-V core demonstrate the effectiveness of coverage-guided aging by achieving a much more regular coverage distribution

A popular verification technique called coverage-guided fuzzing employs mutation-based algorithms to generate new inputs. Despite the great popularity and success story in the software domain, the utilization in the hardware domain is much more limited. In [8], we proposed to leverage *Coverage Guided Fuzzing* (CGF) techniques for cross-level processor verification at the RTL. The approach is guided by the coverage of the reference ISS and the core under test, and uses enhanced custom mutation procedures for common instruction pattern generation. Our fuzzing methodology revealed several bugs in the well-tested RTL processor VexRiscv [9] and thus demonstrated its applicability for processor verification.

While fuzzing is an effective technique, it is still susceptible to miss corner case bugs because it is an inherently incomplete testing approach. Another promising technique of the SW domain is the utilization of symbolic execution. It is a formal verification technique that uses symbolic expression to represent concrete values. Thus, it enables the exploration of large state spaces more efficiently and comprehensively than fuzzing [10]. In [11], we proposed to leverage symbolic execution techniques for cross-level processor verification at the RTL. As a case study, we presented results on the verification of the VexRiscv processor as well as the processor of the open source RISC-V µRV32 (MicroRV32) platform [12]. µRV32 proposes an accessible cross-level platform, providing an FPGA and ASIC compatible RTL description alongside a corresponding binary compatible *Virtual Prototype* (VP). The corresponding VP is built as a configuration of the open source RISC-V VP[1]. The processor of the µRV32 offers configurable support of the RISC-V ISA extensions M (for multiplication/division) and C (for compressed instructions). The SpinalHDL-based RTL description of the µRV32 processor uses the corresponding VP as a reference. For our cross-level processor verification case studies, we used the ISS from the open source RISC-V VP as our functional reference model. Our symbolic verification methodology

[1] See `http://www.informatik.uni-bremen.de/agra/projects/risc-v/` for more information on our RISC-V related work.

revealed several bugs in both RTL cores and thus as well demonstrated its applicability for processor verification.

3 Symbolic Execution for Embedded Software

Symbolic execution is an emerging dynamic software verification technique to enumerate reachable execution paths through a program. The technique has shown success in the conventional domain where it is being employed to uncover critical bugs in software for conventional operating systems like Linux or Windows [13, 14]. Unfortunately, this prior work on symbolic execution is not applicable to the embedded domain, as it does not address challenges specific to the embedded domain:

1. Heterogeneous Ecosystem: The software ecosystem for embedded devices is much more heterogeneous than the ecosystem for the conventional domain. There is a huge variety of different operating systems with different core characteristic and input interfaces which needs to be supported by a symbolic execution engine that is specifically tailored to the embedded domain.

2. Peripheral Modeling: Embedded software interacts on a low abstraction level with peripherals provided by the utilized hardware platform. These interactions need to be supported by the symbolic execution engine in order to comprehensively test the software.

3. Error Detection: Common protection mechanisms against the exploitation of software errors (e.g. those based on hardware peripherals like MMUs) are not widely available on embedded devices. As prior work has shown, many errors therefore remain unnoticed when testing embedded devices using automated software testing techniques [15].

In order to address these challenges, we propose SYMEX-VP, a symbolic execution engine which is specifically tailored to the embedded domain. SYMEX-VP leverages SystemC-based virtual prototypes to provide an executable model of the SiFive1 HiFive1, a RISC-V based microcontroller. As such, SYMEX-VP can execute any RISC-V software targeting the HiFive1 and is thus capable of supporting a diverse software ecosystem. Furthermore, SYMEX-VP is tightly integrated with SystemC [16]. SystemC is a library for the C++ programming language which enables modelling of hardware on a high abstraction level using C++. It is widely used in both academia and industry, and therefore a variety of SystemC models for different hardware peripherals exist already. Through its tight integration with SystemC, SYMEX-VP is capable of supporting low-level peripheral interaction and can even support custom peripherals. Additionally, SymEx-VP supports injecting symbolic test inputs into software simulation through a SystemC TLM extension mechanism by injecting the test inputs through the MMIO peripheral interfaces. This enables the verification of embedded software as-is, with no software modification for the injection of test inputs. SYMEX-VP is further described in [17].

We have used SYMEX-VP as a framework for researching several challenges related to the verification of embedded software. Most importantly, we have proposed several techniques for automatically uncovering spatial memory safety issues in embedded software. For this purpose, we have leveraged prior work on HardBound [18] which enforces spatial memory safety in hardware through a custom peripheral. We have implemented HardBound using SystemC and an LLVM compiler pass to combine it with symbolic execution and thereby employed it as an error detection technique for the embedded domain. This work has been presented in [19]. We have employed this technique to uncover several previously unknown bugs in the popular Internet of Things operating system RIOT [20].

In this regard, we have focused mostly on verifying the network stack of the aforementioned operating system as—in accordance with prior work—we believe it to be the biggest attack vector. Unfortunately, employing symbolic execution to test stateful network protocol implementations is challenging. Symbolic execution is—similar to other dynamic software testing techniques—subject to state space explosion, as the number of paths through the program grows exponentially with the number of branches in the code. This problem is known as *state space explosion* and a well-known limitation of symbolic execution engines. Due to state space explosion issues, it is often unfeasible to explore the entirety of a tested program using symbolic execution. Instead, symbolic execution is often performed within a certain time budget. As such, it is important that deeper, more interesting, parts of the tested code are reached within that time budget to ensure that critical bugs are not missed. To achieve this goal, we combined symbolic execution with manually created protocol specifications in prior work [21, 22]. This allowed us to uncover previously unknown bugs in the MQTT-SN implementation of the RIOT operating system. In summary, the experiments performed in prior work indicate that SYMEX-VP is a capable framework for finding bugs in complex embedded software like the RIOT operating system.

4 Conclusion

Based on our prior work, we have outlined verification techniques, which cover both hardware and software aspects in this extended abstract. Together with a functional reference model of a processor, techniques like CGF and symbolic execution provide powerful processor verification methods. Software that is executed on embedded sys-

tems relies on the interleaved interaction with the hardware. Through our symbolic execution approach we enable early verification of embedded software in its native environment. For this reason, the presented techniques are well-suited for verifying embedded systems which consist of complex software as well as hardware components that need to be comprehensively tested prior to their deployment.

5 References

[1] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, Dec. 2019, Document Version 20191213.

[2] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, Dec. 2021, Document Version 20211203.

[3] ——, "RISC-V ISA tests," https://github.com/riscv/riscv-tests.

[4] "RISC-V Compliance Task Group," https://github.com/riscv/riscv-compliance.

[5] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, "Constrained Random Verification for RISC-V: Overview, Evaluation and Discussion," in *MBMV 2021; 24th Workshop*, 2021, pp. 1–8.

[6] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler, "Efficient cross-level testing for processor verification: A RISC-V case-study," in *FDL*, 2020.

[7] N. Bruns, V. Herdt, E. Jentzsch, and R. Drechsler, "Cross-Level Processor Verification via Endless Randomized Instruction Stream Generation with Coverage-guided Aging," in *DATE*, 2022.

[8] N. Bruns, V. Herdt, D. Große, and R. Drechsler, "Efficient Cross-Level Processor Verification using Coverage-guided Fuzzing," in *GLSVLSI*, 2022, pp. 97–103.

[9] "VexRiscv," 2018, accessed: 2022-07-14. [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[10] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, 2018.

[11] N. Bruns, V. Herdt, and R. Drechsler, "Processor Verification using Symbolic Execution: A RISC-V Case-Study," in *DATE*. IEEE, 2023, accepted.

[12] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, "MircoRV32: an open source RISC-V cross-level platform for education and research," in *Destion '21*, 2021, pp. 30–35.

[13] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *OSDI*, 2008, pp. 209–224.

[14] Vitaly Chipounov and Volodymyr Kuznetsov and George Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, 2011, pp. 265–278.

[15] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *NDSS 2018*, ser. NDSS, San Diego, California, Feb. 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf

[16] System C Standardization Working Group, "IEEE Standard for Standard SystemC Language Reference Manual," IEEE, Tech. Rep., 2012.

[17] S. Tempel, V. Herdt, and R. Drechsler, "SymEx-VP: An open source virtual prototype for OS-agnostic concolic testing of IoT firmware," *JSA*, p. 12, 2022.

[18] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "HardBound: Architectural support for spatial safety of the C programming language," in *ASPLOS*, ser. ASPLOS XIII. New York, NY, USA: Association for Computing Machinery, 2008, p. 103–114.

[19] S. Tempel, V. Herdt, and R. Drechsler, "Automated Detection of Spatial Memory Safety Violations for Constrained Devices," in *ASP-DAC*, ser. ASP-DAC '22, 2022.

[20] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, Dec. 2018.

[21] S. Tempel, V. Herdt, and R. Drechsler, "SISL: Concolic testing of structured binary input formats via partial specification," in *ATVA*, A. Bouajjani, L. Holík, and Z. Wu, Eds. Cham: Springer International Publishing, 2022, pp. 77–82.

[22] ——, "Specification-based Symbolic Execution for Stateful Network Protocol Implementations in the IoT," *IEEE Internet of Things Journal*, pp. 1–1, 2023.