University of Bremen
28359 Bremen, Germany
Group of Computer Architecture
Faculty of Mathematics and Computer Science

DOCTORAL THESIS

Accurate Binary-Level Symbolic Execution of Embedded Firmware

Sören Tempel

June 14, 2024
Date of the Defense

First Examiner:	Prof. Dr. Rolf Drechsler
Second Examiner:	Prof. Dr. Emmanuel Baccelli

Revised version with modifications outlined in Appendix B.

Abstract

Symbolic execution is an automated software testing technique that has enabled the discovery of numerous bugs in conventional, non-embedded software. Unfortunately, its application to embedded firmware is presently limited due to unique challenges associated with this domain. Central to many of these challenges is the tight integration of hardware and software components. Due to this tight integration, firmware interacts on a low abstraction level with both the processor and the peripherals provided by a hardware platform. In order to support these interactions, a symbolic execution engine needs to implement the reference manuals specifying processor and peripheral behavior. These specifications have an enormous complexity; hence, prior work approximates peripheral behavior and abstracts processor instruction execution. Unfortunately, these approximations may induce inaccuracies, which can result in bugs being missed in the tested firmware. This dissertation accomplishes a more accurate analysis by contributing a binary-level symbolic execution approach that is faithful to both the specification of peripheral behavior and processor instruction execution. This is achieved by facilitating machine-readable formal descriptions of instruction semantics and an established modeling standard for the description of peripheral behavior. Conducted experiments have resulted in the discovery of 16 previously unknown bugs in a popular embedded operating system, thereby illustrating the effectiveness of the proposed approach.

Acknowledgements

I would like to thank Rolf Drechsler for creating a great atmosphere for conducting scientific research and guiding me throughout the process. Furthermore, I am grateful to Vladimir Herdt for supervising early work on this thesis and to Christoph Lüth for supporting me in the final years.

Additionally, I would like to thank my current and former colleagues from the Group of Computer Architecture and the Cyber Physical Systems DFKI department. In particular, Tobias Brandt for contributions to the formal RISC-V model presented in this thesis, as well as Pascal Pieper and Sallar Ahmadi-Pour for in-depth discussions and input. Special thanks to colleagues and friends of mine who provided valuable feedback on various early drafts of this thesis, especially Lorenz Hüther and Tristan Bruns.

Last, but not least, I would like to express my gratitude for friends and family supporting me through the years of my studies. In particular, I thank my partner Julia Hunter for her continuous support that helped me immensely to complete this thesis.

Die Universität, deren Bestimmung dem inhumanen Betrieb der Massengesellschaft genau entgegengesetzt ist, muß sich mit aller Macht dagegen wehren, zu dessen Vorhalle zu werden.

Max Horkheimer

Contents

Chapter 1.

Introduction

An embedded system performs computational tasks as part of a larger enclosing product, e.g. a vehicle or an industrial control system [121, Definition 1.1]. Contrary to general-purpose computer systems (such as personal computers), embedded systems can directly cause harm in the physical world if they malfunction. Such malfunctions are often caused by bugs in the software that performs the computational tasks of the embedded system [135]. Some of these software bugs may turn into vulnerabilities that impact important security goals. For example, a logic bug could allow for a denial-of-service attack, thereby impacting the availability of the system [75, Section 3.1.5]. Depending on the enclosing product of the embedded system, this can have severe consequences and—in the worst case—put human lives at risk.

For this reason, it is paramount to prevent the occurrence of such security vulnerabilities. The importance of achieving this goal has become even more relevant in recent years with the emergence of the Internet of Things (IoT) [97]. In the IoT, embedded systems collect information about their surrounding environment and share this information with other systems to reach a common objective [121, Definition 1.3]. Therefore, while it was previously uncommon to be able to directly exchange data with them, embedded systems are becoming increasingly interconnected. This expands the attack vector and affects their threat model, as nowadays it must be assumed that an attacker can communicate with an embedded system over a network connection [75, Section 16.1.1]. As such, the system's input handling routines must be capable of processing arbitrary input received via such a network connection. However, given the complexity of network protocols used in this domain, these input handling routines are susceptible to software bugs and vulnerabilities [159]. The existence of such vulnerabilities in embedded IoT systems is especially critical considering the fact that common protection mechanisms, which attempt to mitigate the exploitation of vulnerabilities, are not widely available on embedded systems [128, 204]. The lack of these protection mechanisms can be attributed to the fact that embedded systems are optimized for characteristics such as chip size, power consumption, or production cost. Compared to conventional general-purpose computer systems, these optimizations lead to severe constraints regarding available computing resources. For example, instead of multiple gigabytes of memory, embedded systems only have a few kilobytes of memory at their disposal [24]. Additionally, hardware features required for the implementation of protection mechanisms (e.g. hardware-enforced isolations) are often unavailable [204, Figure 3]. Therefore, we cannot rely on techniques that mitigate the exploitation of security vulnerabilities during production deployment of an embedded device. Instead, it must be ensured that such vulnerabilities are found and fixed before the embedded system is deployed in a production environment, where undetected vulnerabilities can have severe consequences.

In order to find vulnerabilities in safety-critical systems, it has become an established practice to appoint security experts who perform audits of the system and conduct manual penetration tests [75, Chapter 13]. However, given the ever-increasing complexity of these systems, the use of automated software testing techniques is gaining popularity among practitioners to discover software bugs that may lead to vulnerabilities. In the following paragraphs, an overview of existing techniques in this regard is provided to establish a broad context of prior work.

Static Analysis The term static analysis refers to a software analysis technique which does not execute the tested software, but instead focuses on analyzing its source code. A variety of static analysis tools for different programming languages have been proposed in prior work [52, 120, 19]. Since these tools do not execute the software, they are comparatively easy to employ on a given code base. However, this also comes with the drawback that these static analysis tools produce false-positives as they cannot reason about reachability, i.e. they report bugs in a program that do not constitute an actual problem [40, Section 2.1]. As such, extra effort is required to investigate which reported problems correspond to real bugs.

Fuzzing Contrary to static analysis, dynamic software testing techniques actually execute the tested software. Therefore, these techniques are not subject to false-positives. Instead, they produce a witness (i.e. an input value) which certifies the existence of a bug and with which the bug can be reproduced. A popular automated dynamic software testing technique is fuzzing [123, 140, 23]. The main idea behind fuzzing is to test the software with randomly generated input values, checking if it emits unintended behavior on these inputs (e.g. crashes) [138]. Early work on fuzzing was conducted in 1990 by

Miller et al. [124]. Nowadays, the technique is widely adopted and has found thousands of bugs in popular open source projects [164]. Unfortunately, as it relies fundamentally on random testing, it does not reason about the structure of the tested program and is thus incapable of satisfying complex input constraints.

Formal Methods In comparison to static analysis and fuzzing, formal methods attempt to mathematically reason about a given software and its desired properties. Prior work subsumes "mathematically based languages, techniques, and tools" for specification and verification under this umbrella term [44]. Examples in this regard include: theorem proving [18], model checking [111], or abstract interpretation [50]. While formal methods such as theorem proving provide strong guarantees, employing them on complex existing systems that were not explicitly designed for their application is challenging. This is especially problematic in the embedded domain, where some components (e.g. device drivers) may be vendor-supplied and hence cannot be modified for testing or verification purposes. As such, formal methods are not well suited for finding vulnerabilities in existing embedded systems.

Symbolic Execution Similar to fuzzing, symbolic execution is an automated dynamic software testing technique. However, compared to fuzzing, it is more formal as per the definition given in the previous paragraph. That is, it takes the program structure into account and thereby enables formal reasoning about execution paths through the tested software. This is achieved by executing the software with symbolic input values, which correspond to a set of possible concrete values at a given point in time during program execution (e.g. all values x with $x \geq 10$). The set is continuously constrained in accordance with the program structure, i.e. the constraints enforced by the program upon its input [10, Section 1]. Based on tracked constraints, it is possible to enumerate reachable execution paths for a symbolic input value by formally reasoning about branches that depend on this value using a Satisfiability Modulo Theories (SMT) [14] solver. For example, an SMT solver query can be constructed to determine if there is a concrete assignment for a symbolic value x so that a given branch condition becomes satisfiable. By employing this technique for every branch, symbolic execution can ideally enumerate all execution paths through the program. Unfortunately, in practice, this is often not feasible due to state explosion issues [10, Section 5]. Symbolic execution was originally proposed in 1975 [27, 104], but was initially limited by SMT solver capabilities and has only gained increased relevance in recent years with advances in SMT solving [33].

From these outlined automated software testing techniques, this thesis explores the use of symbolic execution for uncovering vulnerabilities in software for embedded systems. In this regard, we deem state explosion and SMT scalability issues to be less of a problem in this domain, as the software complexity is inherently limited by the constraints of utilized devices (e.g. available memory). While symbolic execution has already been successfully employed to test software for non-constrained conventional devices [32, 41, 61], its application to the embedded domain is presently limited [54, 49, 88]. This is due to unique challenges associated with this domain, which will be discussed in the following.

1.1. Challenges

There is a difference between software for general-purpose computer systems and embedded systems. Prior work uses the term *firmware* for the latter to distinguish the two [68, 122, 210, 45, 54]. Due to the constraints of devices used in the embedded domain, conventional operating systems (e.g. Linux) cannot be used on these devices. Instead, a variety of operating systems specifically tailored to this domain exist [79]. Some embedded devices are even programmed "bare-metal" without using any operating system at all. Therefore, compared to conventional devices, the ecosystem in the embedded domain is much more heterogeneous. Since symbolic analysis presupposes execution of the tested software, it needs to support this diverse ecosystem. Prior work focuses on symbolic execution of conventional software running in user space (instead of kernel space); this is achieved by relying on the homogeneous interfaces used by this software, which are standardized through POSIX [95].[1] In the embedded domain, there typically is no distinction between kernel and user space and the few embedded operating systems (e.g. Tock [114]) that implement such a distinction do not implement a standardized interface between the two. Therefore, symbolically executing firmware for the embedded domain also requires supporting low-level interactions with the utilized hardware platform, which—in the conventional domain—are normally encapsulated by the kernel. These interactions are performed via architecture-specific instructions, e.g. to configure interrupt handlers. A large body of prior work in the symbolic execution domain operates directly on source code [162, 31] or an intermediate representation of this source code [32, 61]. In both cases, it is challenging to support architecture-specific low-

[1] POSIX is a standardized operating system interface specification which is implemented by the majority of general-purpose operating systems for conventional devices like Linux, BSD, or macOS.

level instructions [49, p. 310]. For accurate execution of these instructions, a symbolic execution engine must instead operate directly on the binary-level [207, Section 4.3]. Additionally, it must be taken into account that embedded systems often utilize custom instructions to achieve domain-specific optimizations [51, Section 2]. Naturally, it must therefore be possible to easily extend the binary-level symbolic analysis to support such custom instructions. Performing symbolic execution on the binary-level requires implementing the Instruction Set Architecture (ISA) specification for a chosen hardware platform. These specifications define the semantics of binary instructions and have an enormous complexity, e.g. 6300 pages for the specification of ARMv8-A [4]. Therefore, implementing them is challenging, especially considering that the implementation must be correct, as otherwise bugs may be missed in the tested software.

Unfortunately, operating at the binary-level comes with its own set of challenges. Most importantly, error detection becomes more difficult. This is due to the fact that crucial information for error detection is lost during the compilation process, e.g. information on types [39]. For example, a buffer overflow is challenging to detect at the binary-level as, without any information about the buffer's size, it just looks like a normal memory access. However, detecting buffer overflows is paramount in the embedded domain as the majority of firmware is written in unsafe programming languages like C/C++ which are subject to these errors [181]. Additionally, as most embedded systems offer little to no protection mechanisms against buffer overflow vulnerabilities, they are also trivially exploitable [204]. The lack of protection mechanisms also affects error detection as the majority of memory corruptions (such as buffer overflows) will occur silently and do not result in an observable crash on most embedded systems [128]. Apart from buffer overflows, error classes also differ in the embedded domain due to the constraints of embedded devices. For example, some embedded devices may be battery-powered [24, Section 4.2]. In this case, it must be ensured that it is not possible to drain this battery via a crafted input which results in excessive computations to be performed. Similarly, memory is severely limited on these devices; therefore, an excessive use of memory would also constitute an error and could easily lead to an exploitable vulnerability like a stack overflow [149]. Reasoning about low-level details such as memory use, power consumption, or timing also requires the symbolic analysis to be performed on the binary-level as, otherwise, reliably reasoning about these properties is challenging.

Lastly, embedded systems are—per definition—integrated with an enclosing product and interact with the environment of this product. These interactions are performed via hardware peripherals (e.g. sensors) which are attached to the embedded system [121, Chapter 3]. The firmware interacts on a low abstraction level with these peripherals, either through custom instructions or Memory-Mapped Input/Output (MMIO). Such environment interactions need to be supported by the symbolic analysis [10, Section 4]. Especially considering that inputs retrieved from hardware peripherals may be attacker-controlled and hence of interest for finding vulnerabilities. Depending on the usage scenario, different embedded systems use different hardware peripherals, and some may even use custom peripherals for specialized use cases. Supporting such a diverse set of peripherals is challenging, especially considering the lack of common abstractions for interacting with them [207, Section 8.1.1]. Prior work attempts to resolve this challenge by approximating peripheral behavior [35, 68, 122, 66]. Unfortunately, approximations make the analysis unsound and subject to false-positives, thereby requiring additional manual effort to identify real bugs. Furthermore, the approximations require driver-specific refinements and are therefore incompatible with the heterogeneous embedded ecosystem. Instead, accurate peripheral models are required that are faithful to the real hardware peripheral and hence do not result in false-positives during automated software testing using symbolic execution. Nevertheless, given the complexity and diversity of these peripherals, creating such accurate peripheral models is an open challenge [207, Section 8.1.1], especially considering that they need to be adapted for symbolic execution (e.g. to operate with symbolic values).

1.2. Research Questions

From the outlined challenges, we can deduce the following research questions regarding an application of symbolic execution to firmware for constrained embedded devices:

RQ1 Is it feasible to create accurate, non-approximated models of hardware peripherals, and is it viable to integrate such peripheral models with symbolic execution?

RQ2 How can we implement a binary-level symbolic execution that can cope with the complexity of modern ISA specifications, and is it feasible to extend this approach to additional instructions?

RQ3 How, despite operating on the binary-level, where crucial information for error detection is no longer available, can we detect bugs in the executed code?

RQ4 Does binary-level symbolic execution scale to real-world applications in the IoT, and what are possible optimizations and heuristics specifically for this domain?

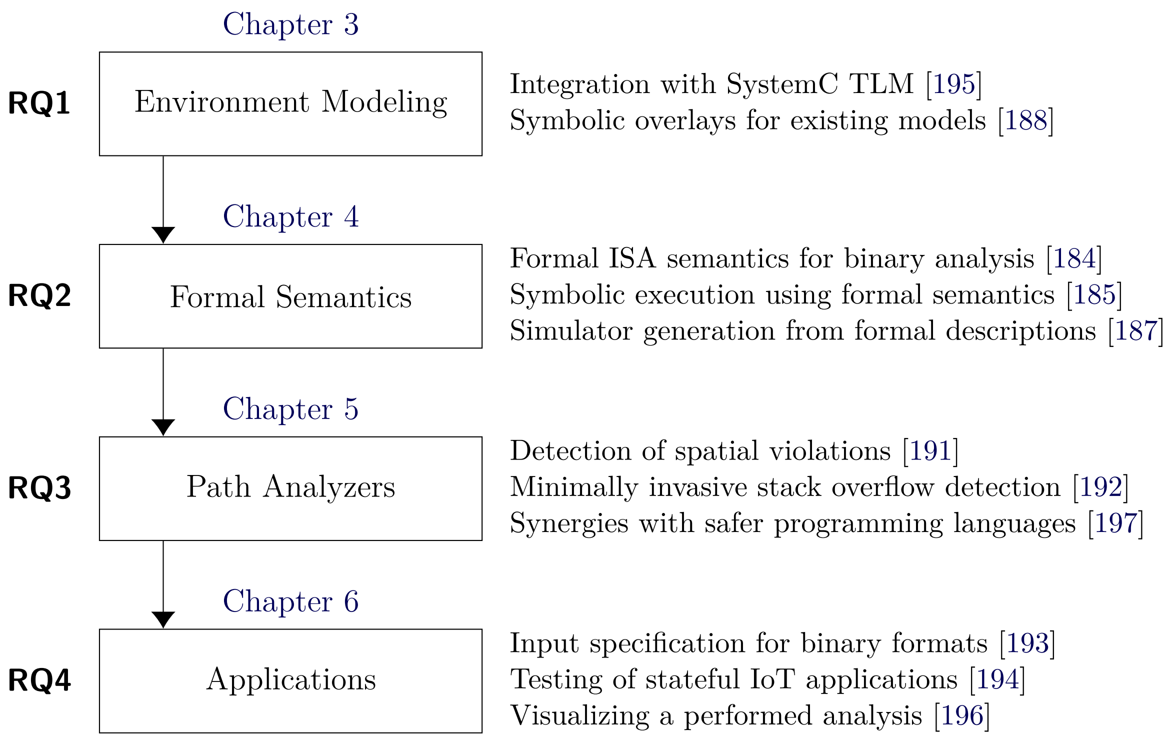

Chapter 3
RQ1
Environment Modeling
Integration with SystemC TLM [195]
Symbolic overlays for existing models [188]
Chapter 4
RQ2
Formal Semantics
Formal ISA semantics for binary analysis [184]
Symbolic execution using formal semantics [185]
Simulator generation from formal descriptions [187]
Chapter 5
RQ3
Path Analyzers
Detection of spatial violations [191]
Minimally invasive stack overflow detection [192]
Synergies with safer programming languages [197]
Chapter 6
RQ4
Applications
Input specification for binary formats [193]
Testing of stateful IoT applications [194]
Visualizing a performed analysis [196]

Figure 1.1.: Contributions of this thesis in relation to the research questions.

1.3. Contributions

Based on the research questions, this section describes the contributions of this thesis. The main contribution is a novel symbolic execution approach that is specifically tailored to firmware for embedded systems. Conducted research on accurate symbolic execution of such firmware has led to the publication of 11 peer-reviewed research articles and research contributions in different domains. The relation between research questions, contribution domains, and published research articles is outlined in Figure 1.1. Based on Figure 1.1, contributions within each domain will be further described in the following.

1.3.1. Contribution Domains

Conceptually, we can distinguish four contribution domains, each of which relates directly to one of the research questions presented in Section 1.2. Contributions within each domain depend on previous research in a prior domain. For example, it is not possible to conduct research on symbolic execution of IoT applications without having accurate models of peripherals utilized in the IoT. Therefore, due to the central role of peripheral modeling, we begin with a description of our contributions in this domain.

Environment Modeling The majority of software is not self-contained; it interacts heavily with its surrounding environment (e.g. to obtain input from a user or to display the result of a performed computation) [10, Section 4]. In the case of embedded firmware, such interactions are performed through peripherals provided by the utilized hardware platform. In order to accurately support these interactions, executable models of these hardware peripherals are required. Hardware can be modeled on different abstraction levels; lower levels offer higher accuracy, while higher levels provide better simulation performance. For dynamic software testing techniques (like symbolic execution), execution speed is essential. Therefore, this thesis facilitates SystemC TLM, a hardware modeling language that operates on a high abstraction level through Transaction-Level Modeling (TLM) [180]. SystemC is standardized by the IEEE and thus well-suited to address the challenges described in Section 1.1, as it enables reuse of existing (vendor-supplied) SystemC models. For similar reasons, SystemC is also a popular choice for the creation of Virtual Prototypes (VPs) [113, Section 2]. A VP provides an executable model of an entire hardware platform, including provided peripherals. In the context of RQ1, this thesis contributes an integration of VPs and SystemC hardware models with symbolic execution. More specifically, a TLM extension for injecting symbolic values into the simulation via the peripheral interface is presented [195]. Additionally, an overlay mechanism is proposed that eases the integration of existing SystemC models with this extension mechanism [188].

Formal Semantics Apart from peripherals, firmware also interacts on a low abstraction level with the Central Processing Unit (CPU) through architecture-specific instructions, for example to implement interrupt handlers. The semantics of these architecture-specific instructions are given by the ISA specification; to accurately support these instructions, a symbolic execution engine needs to be faithful to this specification [207, Section 4.3]. In order to cope with the complexity of modern ISAs, we make use of formal descriptions of ISA specifications that describe the ISA in formal instead of natural language. Contrary to natural language specifications, formal language specifications are explicitly designed to be unambiguous and therefore machine-readable [150, 5, 26, 161, 22]. As such, they can, for example, be used to automatically generate simulators for a given ISA. These generated simulators can then be easily extended to support custom instructions, as long as these instructions can be described in terms of the existing formal language. With regard to RQ2, we contribute a novel executable formal model which is specifically tailored to the creation of binary analysis tools as custom ISA interpreters [184]. Based on this executable formal model, we propose a novel symbolic execution approach which leverages the language primitives of this formal model as an abstraction layer for binary-level symbolic execution [185]. Lastly, we illustrate that such formal models of instruction semantics can also be integrated with VPs through code generation [187].

Path Analyzers The contributions in the prior domains enable the exploration of embedded firmware using symbolic execution. Conceptually, a symbolic execution engine enumerates reachable execution paths through a program based on a specific input source. Additionally, it is necessary to classify and analyze these execution paths (e.g. to determine if a path violates a desired property). Such techniques are known as path analyzers in the symbolic execution domain [41]. In the context of RQ3, we contribute a path analyzer for error detection in embedded firmware. As the majority of firmware is written in the C/C++ programming language [79, Table 1], which does not provide memory safety, we focus on violations of spatial memory safety (e.g. buffer overflows) in this regard [191]. Since protection mechanisms, which mitigate the exploitation of buffer overflows, are not widely available on embedded devices, it is paramount to discover them early as they are trivially exploitable. Prior work has also proposed converting existing C/C++ code to safer languages to improve firmware security [65, 47, 114]. To this end, we contribute a methodology which makes use of these safer languages to ease the detection of errors in embedded firmware [197]. Lastly, we also propose a path analyzer for estimating stack memory usage of embedded firmware. Using this analyzer, we can detect stack overflows in firmware without any instrumentation of the firmware [192].

Applications Based on the prior contributions in the presented domains, we can now employ our symbolic execution approach to test complex real-world firmware to conduct research on RQ4. For this purpose, we focus on an application of our symbolic execution approach to firmware in the IoT. In this domain, the main attack vector is the network stack, which provides an implementation of complex network protocols such as MQTT-SN [176]. In order to test such protocol implementations, we contribute a specification language for specifying partially symbolic message formats of network protocols [193]. Based on an extended version of this specification language, we conduct extensive experiments with stateful network protocol implementations provided by the RIOT and Zephyr IoT operating systems, as part of which we discovered three previously unknown bugs [194]. Finally, we also present a visualization for conducted experiments which enables refinements of created message format specifications [196].

1.3.2. Published Results

The outlined research in the four contribution domains has led to the publication of research results in the form of peer-reviewed articles, presentations at workshops, uncovered bugs in tested software, as well as open source software and evaluation artifacts.

Research Articles In total, the outlined research work resulted in the publication of 11 research articles. As visualized in Figure 1.1, the work presented in Chapter 3, Chapter 4, Chapter 5, and Chapter 6 has been previously published in the following peer-reviewed scientific venues:

[184] Sören Tempel, Tobias Brandt, and Christoph Lüth. "Versatile and Flexible Modelling of the RISC-V Instruction Set Architecture." In: *Trends in Functional Programming*. Ed. by Stephen Chang. Boston, MA, USA: Springer International Publishing, Jan. 2023, pp. 16–35. ISBN: 978-3-031-21314-4. DOI: `10.1007/978-3-031-38938-2_2`.

[187] Sören Tempel, Tobias Brandt, Christoph Lüth, and Rolf Drechsler. "Minimally Invasive Generation of RISC-V Instruction Set Simulators from Formal ISA Models." In: *2023 Forum on Specification & Design Languages*. FDL. Turin, Italy, Sept. 2023, pp. 1–8. DOI: `10.1109/FDL59689.2023.10272224`.

[188] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "An Effective Methodology for Integrating Concolic Testing with SystemC-based Virtual Prototypes." In: *2021 Design, Automation & Test in Europe Conference & Exhibition*. DATE. Grenoble, France, Feb. 2021, pp. 218–221. DOI: `10.23919/DATE51398.2021.9474149`.

[191] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "Automated Detection of Spatial Memory Safety Violations for Constrained Devices." In: *Proceedings of the 27th Asia and South Pacific Design Automation Conference*. ASPDAC. Taipei, Taiwan, Jan. 2022, pp. 160–165. DOI: `10.1109/ASP-DAC52403.2022.9712570`.

[192] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "In-Vivo Stack Overflow Detection and Stack Size Estimation for Low-End Multithreaded Operating Systems using Virtual Prototypes." In: *2021 Forum on Specification & Design Languages*. FDL. Antibes, France, Sept. 2021, pp. 1–7. DOI: `10.1109/FDL53530.2021.9568384`.

[193] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "SISL: Concolic Testing of Structured Binary Input Formats via Partial Specification." In: *Automated Technology for Verification and Analysis.* Ed. by Ahmed Bouajjani, Holík Lukáš, and Zhilin Wu. ATVA. Beijing, China: Springer International Publishing, Oct. 2022, pp. 77–82. ISBN: 978-3-031-19992-9. DOI: `10.1007/978-3-031-19992-9_5`.

[194] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "Specification-Based Symbolic Execution for Stateful Network Protocol Implementations in IoT." In: *IEEE Internet of Things Journal.* IoT-J 10.11 (Jan. 2023), pp. 9544–9555. DOI: `10.1109/JIOT.2023.3236694`.

[195] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "SymEx-VP: An Open Source Virtual Prototype for OS-agnostic Concolic Testing of IoT Firmware." In: *Journal of Systems Architecture.* JSA (May 2022), pp. 1–12. ISSN: 1383-7621. DOI: `10.1016/j.sysarc.2022.102456`.

[196] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "Towards Quantification and Visualization of the Effects of Concretization During Concolic Testing." In: *IEEE Embedded Systems Letters.* ESL 14.4 (Dec. 2022), pp. 195–198. DOI: `10.1109/LES.2022.3171603`.

[197] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "Towards Reliable Spatial Memory Safety for Embedded Software by Combining Checked C with Concolic Testing." In: *2021 58th ACM/IEEE Design Automation Conference.* DAC. San Francisco, California, Dec. 2021, pp. 667–672. DOI: `10.1109/DAC18074.2021.9586170`.

Additionally, the following research article has been submitted to the *International Conference on Software Engineering and Formal Methods* (SEFM'24), but is—at the time of writing—still under review and hence unpublished:

[185] Sören Tempel, Tobias Brandt, Christoph Lüth, and Rolf Drechsler. "Accurate and Extensible Symbolic Execution of Binary Code based on Formal ISA Semantics." In: *International Conference on Software Engineering and Formal Methods.* SEFM. Aveiro, Portugal, Nov. 2024, pp. 1–18. Under Review.

Miscellaneous Apart from the outlined articles, research results have also been presented at five workshops. Additionally, performed experiments have led to the discovery of 16 previously unknown bugs in the popular RIOT [9] operating systems. In order to allow researchers and practitioners to employ symbolic execution for testing their embedded systems, all tooling developed as part of the conducted research has been released as open source software. Furthermore, whenever possible, artifacts for performed evaluations have also been published as part of the research results. Both open source repositories and evaluation artifacts are referenced in the corresponding chapters.

1.4. Outline

This thesis starts with a preliminaries chapter (Chapter 2) which provides background information on key topics and technologies that form the basis of the research conducted in this thesis. This includes an introduction to the design and development of embedded systems in the context of electronic design automation. The introduction focuses on the modeling of these systems using established methods such as virtual prototyping and SystemC. This introduction is followed by a discussion of ISA specifications. The ISA mandates the interface between the hardware and the software of a computer system. Given the close integration of software and hardware in the embedded domain, this interface specification is central for testing the firmware of embedded systems. Additionally, Chapter 2 provides a more detailed, example-driven introduction to symbolic execution.

In the following chapters, the contributions of this thesis are developed alongside the contribution domains as outlined in Figure 1.1. The first chapter in this regard (Chapter 3) focuses on environment modeling; to this end, it presents an integration of SystemC hardware peripheral models with symbolic execution. The following chapter (Chapter 4) improves upon this integration by leveraging formal models of ISA semantics to ease extending the analysis to different embedded systems. Based on the symbolic execution engine developed in these chapters, multiple path analyzers are discussed which enable error detection in firmware for embedded systems (Chapter 5). Finally, different real-world applications are tested using the developed techniques. In order to improve input generation for this use case, domain-specific heuristics and optimizations are presented (Chapter 6).

The thesis is concluded in Chapter 7. In this chapter, results with regard to the outlined research questions are summarized, and potential opportunities for future work are laid out.

Chapter 2.

Preliminaries

This chapter provides preliminary background information on key topics and technologies of this thesis and serves as a prerequisite for the following chapters. This includes an introduction to the design and development of embedded systems, ISA specifications, and symbolic execution.

2.1. Embedded Systems

As defined in Chapter 1, an embedded system performs computational tasks as part of a larger enclosing product [121, Definition 1.1]. The complexity of these systems is continuously increasing, especially with the recent emergence of the IoT. In the IoT, virtual things (i.e. software) collect information on physical things (i.e. the surrounding physical environment) via sensor peripherals provided by the embedded system. The vision behind the IoT is to build a new "global infrastructure for the information society" that enables advanced services by interconnecting physical and virtual things [97]. Realizing this vision requires building large networks of interconnected embedded devices. In order to make building such networks financially viable, the characteristics of each device used in this domain must be scaled down to reduce their production cost and physical size. This causes these devices to be severely constrained in terms of available computing resources (e.g. available memory) compared to conventional desktop devices. For example, instead of multiple gigabytes of memory, these devices only have kilobytes of memory at their disposal. Prior work by Bormann et al. on RFC 7228 therefore introduces the term *constrained devices* to refer to devices of this kind and presents a classification of these devices according to energy sources and available memory [24]. The latter classification is shown in Table 2.1 where three device classes from RFC 7228 are presented. These classes differ in the severity of their memory constraints; in this regard, Table 2.1 distinguishes between available code and data size. The most constrained device class is *C0*, the least constrained device class is *C2*. However, even devices of the least constrained

Table 2.1.: Classification of constrained devices from RFC 7228 [24, Table 1].

Device Class	Data Size	Code Size
Class 0 (C0)	Less than 10 KiB	Less than 100 KiB
Class 1 (C1)	Around 10 KiB	Around 100 KiB
Class 2 (C2)	Around 50 KiB	Around 250 KiB

device class only have access to roughly 50 KiB of data size and 250 KiB of code size. Therefore, conventional software and conventional operating systems (such as Linux) cannot be used on these devices. Instead, an entire ecosystem of operating systems and software optimized for these constraints exists [79].

The constraints also complicate the design of new embedded systems, as it must be ensured that software components are compatible with hardware constraints. This is further amplified by the importance of the *time-to-market*, i.e. the time span from the initial conception of a product to its availability on the consumer market, which must be as short as possible. As such, it is paramount to reduce the development time of embedded systems and to facilitate early development and evaluation of software components in order to examine their compatibility with hardware constraints. These aspects must already be considered during the design process of an embedded system. As illustrated in Figure 2.1, a traditional design process first conceives and implements the hardware, and then—as soon as the hardware becomes available—the development of software components begins. This traditional design flow has two problems: it results in a longer time-to-market as software can only be developed after hardware development is completed, and software cannot be evaluated early, making it challenging to adjust or refine the hardware based on such evaluations (e.g. equip the hardware with more memory). In order to overcome these challenges, it is becoming increasingly popular to leverage VPs for the development of new embedded systems [160, Section 1]. A VP provides an executable model of an entire hardware platform, including the provided peripherals. This allows software development to begin before the physical hardware becomes available, thereby reducing the time-to-market and enabling early evaluations of software components [160, Section 1.3]. This is illustrated on the right-hand side of Figure 2.1. In a VP-based design flow, the VP is developed first and then, based on

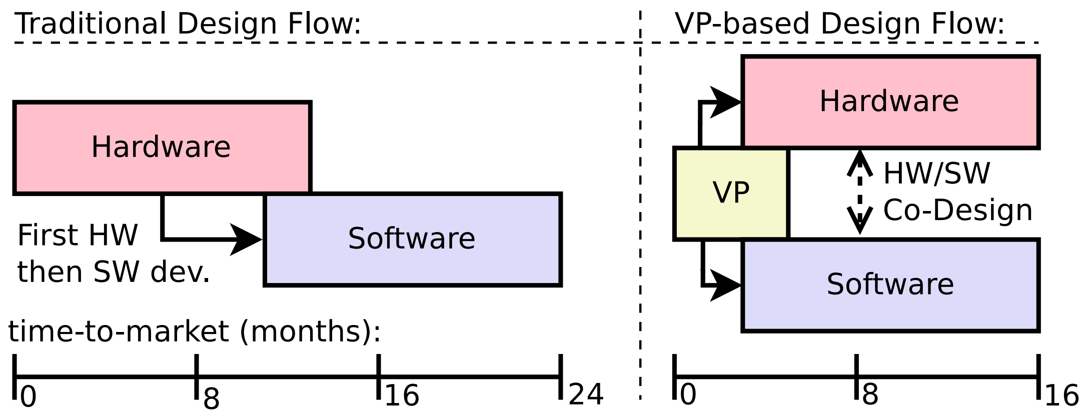


Figure 2.1.: Illustration of a VP-based hardware design flow [82, Figure 1.1].

the VP, software and hardware components are developed in parallel. This allows for a significant reduction of the time-to-market.

In order to enable early software development, a VP needs to provide executable models of hardware peripherals. For example, models of sensors that are used to collect information on the surrounding physical environment of an embedded system. Without such executable models, software interacting with these peripherals cannot be developed and tested during the early stages of the development process. Within the VP domain, such peripheral models are predominantly created in SystemC [180], a C++ class library for modeling hardware systems. More specifically, this thesis uses SystemC TLM, which models peripherals on a high abstraction level using a bus abstraction [100]. Interactions with peripherals are modeled by exchanging TLM transactions over this bus abstraction, e.g. to read or write a register of the modeled peripheral. Compared to standard hardware description languages like VHDL [59] or Verilog [58], which operate on the Register-Transfer Level (RTL), the higher abstraction provided by TLM eases the creation of new peripheral models and allows for better simulation performance. This is beneficial for an integration with symbolic execution, as simulation performance is a vital property in this domain to mitigate state explosion issues, i.e. explore as much of the software as possible in as little time as possible.

2.2. Instruction Set Architectures

Apart from hardware models, a VP also needs to implement the ISA of a chosen hardware platform. An ISA specification covers different aspects, including the instructions of a processor, its state (number and types of registers), and its memory model. As

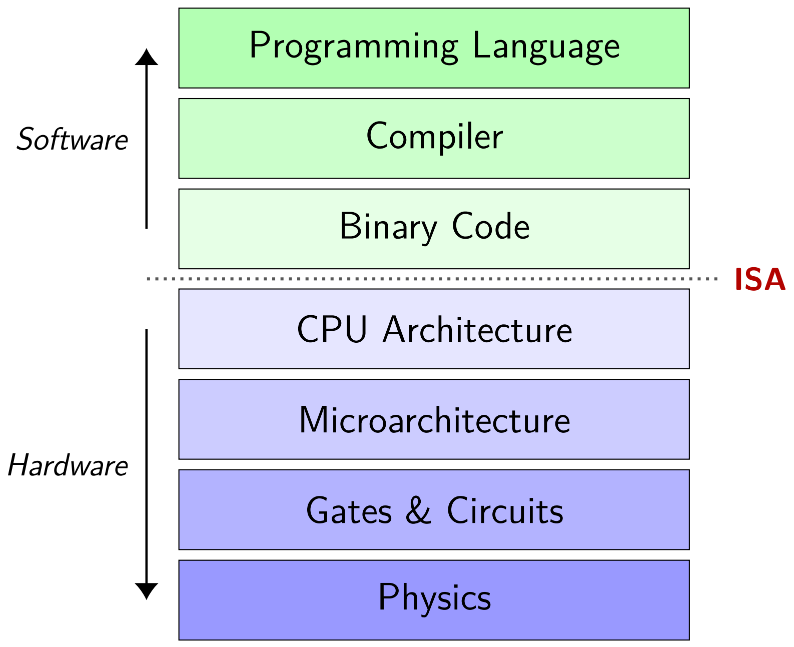


Figure 2.2.: Relation of the ISA to other software and hardware abstraction layers.

illustrated in Figure 2.2, the ISA is therefore the central interface between the hard- and software and conceptually forms the boundary between the two. Software in high-level programming languages (e.g. C/C++) is translated by a compiler to binary code, which then uses the instructions of a specific ISA (e.g. x86). These instructions are then implemented in hardware by the CPU. In order to execute software in binary form, it must first be loaded into memory. Afterward, the CPU fetches one instruction at a time from memory, decodes it, and then executes it. This process is known as the *fetch-decode-execute cycle* [175, Section 14.3].

As per Chapter 1, this thesis proposes direct symbolic execution of embedded firmware in binary form. In order to execute binary code firmware symbolically, we need to implement symbolic semantics for the binary code instructions mandated by the utilized ISA. Additionally, we need to ensure that instruction operands (registers or memory values) can be represented as symbolic values. For this purpose, we need to build a custom Instruction Set Simulator (ISS). Naturally, the implementation of such a simulator is ISA-specific, as different ISAs mandate different instructions with different semantics. A variety of different ISAs exist; popular examples include ARM, MIPS, x86 or IBM Power. In recent years, a new ISA, called RISC-V [153, 154], has gained popularity in both industry and academia. Contrary to the aforementioned examples, RISC-V is developed as an open standard free from patents and royalties. Initially, the ISA was developed at the University of California but is now managed by the non-profit RISC-V International organization (formerly known as RISC-V Foundation). Due to its openness, RISC-V serves as the basis for a lot of research involving ISAs and for this reason, it is also extensively used in the contribution domains of this thesis. Furthermore, RISC-V is a modular ISA, meaning the specification consists of a mandatory base instruction and optional extensions that can be implemented on top and combined as needed. As such, the specification is constantly expanding through the standardization of new instruction set extensions. This requires developers of binary analysis tools to "catch up" by implementing these new extensions. Since this is a manual error-prone process, RISC-V benefits from an extensible symbolic execution approach that can easily support new instruction set extensions. RISC-V's modularity also allows it to scale to different use cases, from embedded devices to desktop hardware and supercomputers. In order to accommodate these different use cases, three configurations of the ISA are available, which differ in regarding the register width. These configurations are known as RV32, RV64, and RV128 and provide 32-bit, 64-bit, and 128-bit register widths respectively. Since this thesis concerns itself with embedded devices, it focuses on the 32-bit configuration of the RISC-V ISA.

2.3. Symbolic Execution

Symbolic execution is a dynamic software testing technique that attempts to enumerate reachable execution paths through a program based on a specific input source. This is achieved by treating variables—corresponding to the input source—as symbolic values. Symbolic values represent a set of possible concrete values, e.g. all values x with $x > 10$. This set is continuously constrained during software execution to match the constraints enforced by the tested program upon its input. For example, if a branch with the condition $x > 12$ is taken, then, from that point onward, the symbolic variable x can only refer to values that are greater than 12. Based on the tracked constraints, it is possible to formally reason about branch points in the program using an SMT solver. That is, we can consult the SMT solver to determine if a branch condition, which depends on a symbolic value x, can be satisfied under the current constraints. Contrary to other dynamic software testing techniques, such as fuzzing, symbolic execution requires custom interpretation of the tested software in order to have it operate on symbolic values. The advantage of this is that it enables symbolic execution to take the program structure into account for input generation and allows it to perform formal reasoning. By formally rea-

```c
int myfunc(int a) {
  int ret = 0;

  if (a > 8) {
    ret = a - 7;
  }

  if (a < 5) {
    ret = a - 2;
  }

  return ret;
}
```

Listing 2.1.: Example code for illustrating symbolic execution of a C/C++ function.

soning about every branch point, we can ideally enumerate all execution paths through a program. Unfortunately, completeness is often not achievable in practice as the number of paths through a program grows exponentially with the number of branches in the tested code (path explosion) [10, Section 5].[1]

Path explosion is commonly referred to as the "biggest challenge facing symbolic execution" [33, p. 87]. In order to mitigate this issue, a variety of symbolic execution algorithms and heuristics have been proposed in prior work [10]. In this thesis, we employ Dynamic Symbolic Execution (DSE), a variant of symbolic execution that is driven by concrete execution. More specifically, we utilize a DSE flavor that mixes concrete and symbolic execution and is hence commonly referred to as *concolic execution* [10, Section 2.1]. Concolic execution concretely explores one path at a time, but tracks constraints on symbolic values and branches that depend on them alongside the execution in a binary tree. Based on this tree, an SMT query can be constructed that, if satisfiable, yields an input value that results in the discovery of a new branch. For example, such a query can be constructed by negating the condition of a branch for which only the true but not the false case has been explored. If the resulting query is satisfiable under the path constraints, then the concrete execution is restarted with the input value returned by the SMT solver. This is best illustrated using an example. Consider the

[1]Path explosion is also commonly referred to as "state explosion". In this thesis, the two terms are used interchangeably.

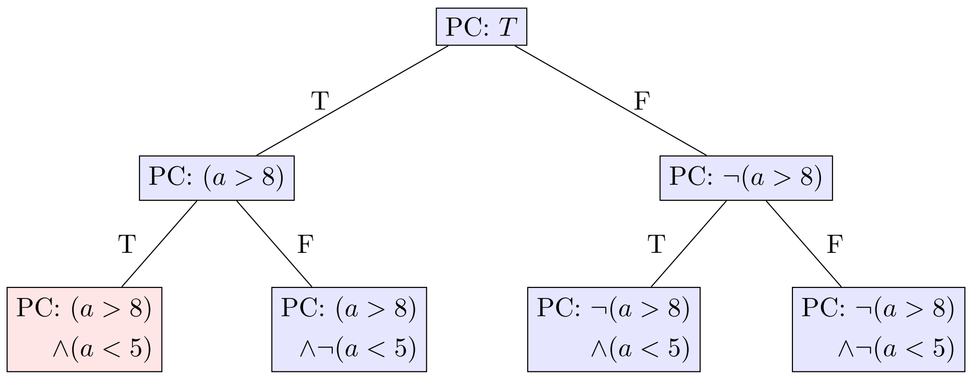

PC: T
T
F
PC: $(a > 8)$
PC: $\neg(a > 8)$
T
F
T
F
PC: $(a > 8)$ $\wedge(a < 5)$
PC: $(a > 8)$ $\wedge\neg(a < 5)$
PC: $\neg(a > 8)$ $\wedge(a < 5)$
PC: $\neg(a > 8)$ $\wedge\neg(a < 5)$

Figure 2.3.: Binary tree for exploration of the `myfunc` function from Listing 2.1.

C/C++ function `myfunc` as defined in Listing 2.1. This function takes a single function parameter of type `int` and has two branches that depend on this parameter. If we want to explore this function, we would need to treat this parameter as an unconstrained symbolic value. For the initial concrete exploration using DSE, the value can therefore be arbitrary (e.g. random). Independent of the concrete value for the initial exploration, we would encounter two branches during this initial exploration, which would be tracked in an execution tree. At the end of execution, we can then negate one of these using an SMT solver and restart execution until all branches have been negated (i.e. all paths have been explored). Figure 2.3 illustrates the final binary tree that would be iteratively built alongside execution. In Figure 2.3, each node corresponds to an executed branch in Listing 2.1 and shows the path constraints (PC) enforced by the taken branch. The path constraints are updated according to encountered branch conditions, depending on whether the branch condition is assumed to be false (F) or true (T), as specified by the edges in Figure 2.3. The nodes in the first level show the path constraints enforced by the branch in Line 4 - Line 6 of Listing 2.1, the nodes in the second level show the path constraints of the branch in Line 8 - Line 10. Each path in Figure 2.3, from the root node to a terminal node, represents an execution path for `myfunc`. The path leading to the terminal node with condition $(a > 8) \wedge (a < 5)$ is not reachable, as the condition is not satisfiable. Therefore, a symbolic execution engine would discover three execution paths through the function `myfunc` based on an unconstrained symbolic function parameter.

In order to track constraints alongside execution, symbolic execution requires custom software interpretation. As per Chapter 1, the symbolic execution proposed in this thesis operates directly at the binary code level. That is, it symbolically executes instructions mandated by the ISA (see Section 2.2). For an implementation of concolic execution, the instruction operands are treated as concolic values: tuples with a concrete and an optional symbolic part. This allows tracking symbolic constraints alongside the concrete execution. Additionally, concolic values allow for efficient concretization (i.e. conversions of symbolic values to concrete ones) by discarding the optionally symbolic part from the tuple. As discussed in Section 2.2, execution and decoding/parsing are heavily intertwined for machine code. Among other things, we make heavy use of concretization for converting fetched memory words to concrete values for instruction decoding; therefore, efficient concretizations are paramount for our symbolic execution approach. In the following chapters, we use the term “symbolic execution” to refer to the general concept and “concolic execution” to refer to a specific instantiation of this concept in the context of our symbolic execution implementation.

Chapter 3.

Integration of SystemC TLM with Symbolic Execution

Any software that performs a meaningful function needs to interact with its surrounding environment, e.g. to obtain input for a computation from a user or to display the results of a performed computation. In order to accomplish these interactions, software relies on the surrounding software stack and provided hardware peripherals, accessed through the abstractions supplied by this software stack. For example, software may obtain input from a hard drive through the file system abstraction provided by a utilized operating system. Dynamic software testing techniques need to support these interactions in order to properly analyze the tested software. Specifically for symbolic execution, the challenge in this regard is that symbolic values need to propagate across the boundaries of interfaces provided for such interactions. As an example, if a program writes symbolic values to a file, a later read of this file should return symbolic values with the same constraints [10, Section 4]. Interactions with the environment, that need to be supported by the symbolic execution engine, depend on the kind of software that is being tested. Prior work on symbolic execution of software for conventional operating systems (e.g. Linux, BSD, macOS) focuses on the interfaces mandated by the POSIX standard [32, 41, 8]. For example, the KLEE symbolic execution engine provides an abstract symbolic model for the POSIX file system interface [32, Section 4.1]. Unfortunately, as explained in Section 1.1, the embedded software ecosystem is much more heterogeneous and no widely adopted interface specification (like POSIX) exists in this domain. Instead, embedded firmware interacts directly with the hardware and the provided peripherals through MMIO and architecture-specific low-level instructions. A symbolic execution engine for embedded firmware needs to support such hardware environment interactions.

While doing so would be possible on real hardware, dynamic software testing techniques which depend on hardware are severely limited by the constraints imposed upon them by the characteristics of constrained embedded devices, e.g. slow execution speed (see Section 2.1). A solution to this problem is emulation of the embedded firmware on a more potent, non-constrained system. However, emulators like QEMU [16], on which dynamic analysis tools such as S^{2}E [41] and FIRMADYNE [37] are based, only offer limited support for peripherals and thus do not support a wide range of embedded firmware images. For this reason, prior work proposed hybrid emulation where code interacting with peripherals is forwarded from an emulator to the real hardware [127, 49, 109]. Unfortunately, constantly transferring the execution state between the emulator and the real hardware results in a performance penalty. An alternative direction pursued in prior work is approximating peripheral behavior, thereby reducing the effort required to correctly model utilized peripherals within an emulator and often achieving device-agnostic firmware execution, but trading this advantage for soundness of the performed analysis (i.e. results can include false-positives) [35, 68, 54]. Additionally, such approaches require driver-specific refinements, which is a laborious process and complicates the application of this technique to a wide range of different firmware images with different device driver implementations. Yet another approach includes targeting specific Hardware Abstraction Layers (HALs) but this approach is only applicable to firmware images using a certain HAL and thus not agnostic [45]. A detailed survey of prior work in this domain is provided by Fasano et al. [66].

The aforementioned survey also acknowledges shortcomings of prior work in regard to environment modeling and suggests the widespread adoption of modeling standards to mitigate them [66, Section 6]. By facilitating such standards, existing accurate and—ideally verified—models of hardware behavior can be reused and integrated with symbolic execution engines for firmware testing purposes. Hardware modeling is its own area of research with a large body of existing work [121, Chapter 2]. A comparatively recent development in this domain are VPs [113]. VPs model the entirety of a hardware platform, including peripherals provided by this platform. As explained in Section 2.1, peripheral models for VPs are predominantly created using the hardware modeling standard SystemC [160, Section 2.5.2]. SystemC [180] is based on the C++ programming language and allows modeling hardware peripherals on a higher abstraction level than traditional Hardware Description Languages (HDLs) such as VHDL [59] or Verilog [58]. This is achieved through TLM, a modeling technique where peripheral interactions are described based on a bus abstraction [100]. Operating on a high abstraction level through TLM eases the creation of peripheral models and also allows achieving high simulation speed, an important characteristic for dynamic software testing techniques such as symbolic execution. For this reason, this chapter investigates the use of TLM in combination with the SystemC standard for modeling interactions with the environment during symbolic execution of embedded firmware.

Section 3.1 presents an integration of symbolic execution with an existing VP for the RISC-V architecture. This integration makes use of a TLM extension mechanism for transporting symbolic values over the bus abstraction. In order to ease the integration of existing standard-conforming SystemC peripheral models with this extension mechanism, Section 3.2 presents an overlay mechanism that achieves this integration without requiring peripheral modifications. Thereby easing the adaptation of vendor-supplied peripheral models for which source code may not be available.

3.1. Combining VPs with Symbolic Execution

In this section, we present SymEx-VP, a symbolic execution engine for agnostic testing of embedded firmware. As the name suggests, SymEx-VP employs a combination of symbolic execution and virtual prototyping, thereby addressing challenges outlined in Section 1.1 regarding the handling of environment interactions during dynamic testing of firmware in the embedded domain. SymEx-VP is based on `riscv-vp`, an existing open source VP for the RISC-V architecture which was initially developed by Herdt et al. in prior work [85]. This VP provides an executable model for hardware platforms such as the SiFive HiFive Unleashed[1] or the SiFive HiFive1[2]. Apart from SystemC TLM peripheral models for these platforms, a central component of any VP is the ISS. As explained in Section 2.1, VPs are intended to facilitate early development of software components for embedded systems. For this purpose, they need to support accurate simulation of software in binary form. In this context, the ISS component of a VP is responsible for executing the instructions mandated by the ISA of the chosen hardware platform. The aforementioned hardware platforms are based on the open standard RISC-V architecture introduced in Section 2.2; thus, the existing `riscv-vp` already provides an ISS for this architecture. Unfortunately, a unique characteristic of symbolic execution is that it requires custom software interpretation (refer to Section 2.3 for details). As such, for the integration of VPs and symbolic execution, the existing ISS had to be modified to support symbolic firmware execution.

In order to integrate VPs with symbolic executions, this section proposes a sym-

[1]`https://www.sifive.com/boards/hifive-unleashed`
[2]`https://www.sifive.com/boards/hifive1`

bolic ISS for the RISC-V architecture and an integration of SystemC TLM hardware models with symbolic execution through a TLM extension mechanism. The proposed contributions are implemented in SymEx-VP and, based on this implementation, case studies are conducted to illustrate that such an integration is beneficial for automated testing of embedded firmware. In fact, to the best of our knowledge, SymEx-VP is the only symbolic execution with support for accurate peripheral models via a dedicated hardware modeling language (SystemC). In order to stimulate future research in this direction, SymEx-VP is available as open source software on GitHub: `https://github.com/agra-uni-bremen/symex-vp`. More information on the integration of VPs with symbolic execution, as well as a description of case studies conducted to evaluate this integration, will be provided in the following sections.

3.1.1. The Case for SystemC TLM

VPs provide an executable virtual model of a physical hardware platform and allow early execution of software targeting this platform. As explained in Section 2.1, they are a popular tool for the development of new embedded systems, as the early creation of VPs enables the development of both hardware and software components in parallel, thereby reducing the time-to-market [160, Section 1.3]. This is achieved by utilizing VPs to simulate the behavior of the targeted hardware platform, thus allowing software development to begin before the physical hardware is available. Since embedded software is tightly integrated with the hardware, simulating execution of the software—which interacts with the hardware—requires accurate models of hardware behavior. As per Section 1.1, such accurate models are also paramount to symbolically execute embedded firmware which interacts with a wide range of different peripherals.

Conceptually, hardware can be modeled on different abstraction levels (e.g. the transaction, register transfer, or circuit level) [121, Section 2.9]. In regard to these different abstraction levels, there is a trade-off between model accuracy and simulation performance. Lower abstraction levels (e.g. RTL) allow for more accurate hardware models but achieve worse simulation performance than higher abstraction levels (e.g. TLM). For both, automated firmware testing and the creation of VPs, we desire high simulation performance [113, Section 1.3]. Therefore, we want to operate on a high abstraction level. For each abstraction level, a variety of languages exists [121, Section 2.9]. When operating on a high level, it is naturally possible to model hardware using general-purposes programming languages such as C++.[3] However, such languages do not provide us with a common Application Programming Interface (API) for creating hardware models, thereby making it difficult to reuse existing models. For this reason, as suggested in prior work, we want to build upon modeling standards to overcome this limitation and enable reuse of existing (ideally verified) peripheral models for automated firmware testing using symbolic execution [66, Section 6]. In an industrial setting, it is then even possible to use vendor-supplied peripheral models for testing purposes.

In this context, SystemC is a good fit as it is a widely adopted modeling standard maintained by the IEEE [180]. The standard itself supports hardware modeling on different abstraction levels. We make use of SystemC TLM in this thesis, which operates on the transaction level [100]. As SystemC is an established standard, there is also a large body of prior work on the verification of SystemC hardware models [110, 78, 42]. This work is complementary to our own as it allows verifying model accuracy, thereby ensuring that no bugs are missed in the tested firmware due to inaccuracies of the utilized hardware models. For these reasons, we contribute an integration of VPs, which utilize SystemC TLM hardware models, with symbolic execution.

3.1.2. SymEx-VP Architecture

In this section, we provide more details on the implementation and architecture of SymEx-VP and illustrate how we modified the existing `riscv-vp` [85] from prior work by Herdt et al. in order to achieve an integration of VPs with symbolic execution.

3.1.2.1. Overview

An overview of the SymEx-VP software architecture is provided in Figure 3.1. Since our architecture is based on SystemC TLM, the central component of our architecture is the TLM bus. Similar to standard SystemC-based VPs, provided peripherals (e.g. sensors) are connected to this bus. The instructions of the firmware executed by the VP are interpreted by the ISS. The firmware executed by the ISS communicates with peripherals attached to the TLM bus via load/store instructions (i.e. MMIO). These instructions are converted by the memory interface to TLM transactions and forwarded to the targeted peripheral via the TLM bus (see Section 2.1). A special `SymbolicCTRL` peripheral is provided to allow the executed firmware to communicate with SymEx-VP,

[3]Though it is challenging to model hardware-level concurrency in these languages [21, Section 1.3.3.2].

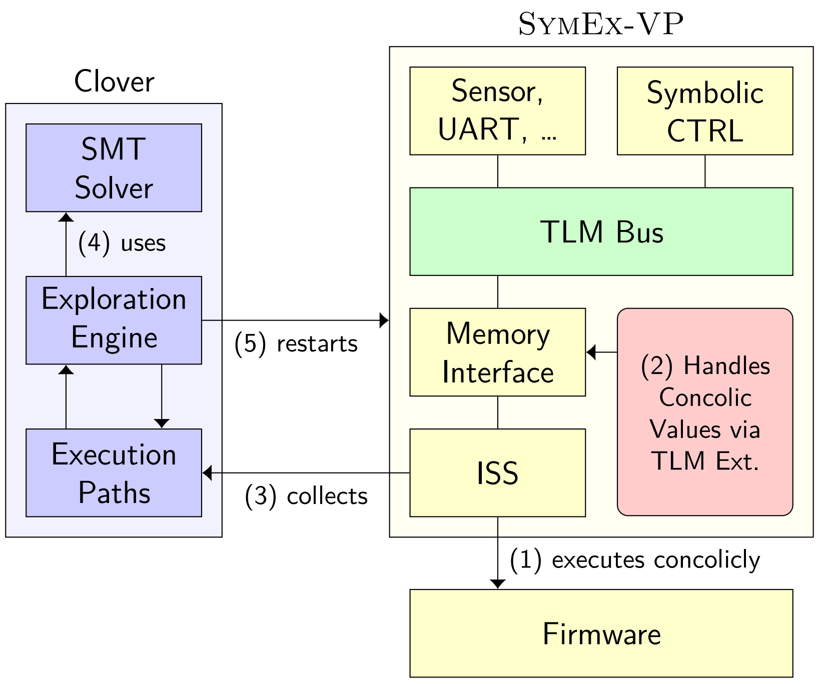


Figure 3.1.: Overview of our Clover-based SymEx-VP software architecture.

thereby allowing the firmware to influence and configure the performed symbolic analysis (see Subsection 3.1.2.5).

SymEx-VP diverts from the standard SystemC TLM architecture by utilizing a modified ISS which operates on symbolic—instead of concrete—values. More specifically, as discussed in Section 2.3, SymEx-VP implements concolic execution and thus operates on concolic values. As such, RISC-V instructions executed by the ISS have concolic operands. These concolic operands originate via performed load/store instructions (i.e. MMIO interactions) in provided peripherals. Peripherals can inject concolic values via the TLM extension mechanism. This mechanism allows transporting arbitrary C++ classes alongside standard TLM transactions, thereby enabling modeling of more complex bus protocols that do more than just copying blocks of data [180, Section 14.2]. We use this mechanism to transport our C++ class for representing concolic values over the TLM bus. TLM transactions with an attached extension are detected by the memory interface, which then extracts the concolic values from the extension and passes them to the ISS. The tested firmware is then executed based on these concolic values, and branches as well as constraints on these values are collected by the ISS and passed to a separate component called Clover. Clover provides an implementation of well-known concolic execution algorithms.

As soon as the SystemC simulation stops (i.e. firmware execution ends), SymEx-VP consults Clover to determine new assignments for concolic values based on the collected branches. For this purpose, Clover iteratively creates an execution tree consisting of collected branch conditions. Based on this execution tree, a DSE algorithm selects a random branch condition from this tree to negate, generates an SMT query for this negated branch condition, and solves this query to generate a new assignment for involved concolic values. The entire SystemC simulation is then restarted by the exploration engine with these new assignments. This process is ideally repeated until all paths through the program have been discovered or a given time budget is exceeded.

3.1.2.2. Clover

In order to implement this proposed architecture on top of the existing `riscv-vp`, the ISS needs to be modified to execute machine code instructions concolicly. As such, the data type used for instruction operands needs to be changed from a concrete to a concolic one. The implementation of these data types and algorithms is not specific to SystemC; therefore, we encapsulated them in an independent concolic testing library we named Clover. The Clover library provides us with concolic data types and implements operations (e.g. addition or subtraction) on these types. Furthermore, it provides a DSE (Section 2.3) implementation and an interface for the Z3 [125] SMT solver. The Clover solver interface for Z3 is based on KLEE, an existing open source symbolic execution engine described further in prior work by Cadar et al. [32]. By re-using the KLEE solver interface, we can benefit from existing KLEE query optimizations—implemented on top of existing SMT solvers—such as the counterexample cache [32, Section 3.3]. Clover is written in roughly 1000 LOC of C++ and includes a slightly modified and stripped-down version of KLEE, it is also freely available on GitHub.[4] We hope that by providing core concolic execution components as a separate library, we can ease the integration of our approach with other existing VPs.

3.1.2.3. Instruction Set Simulator

Based on the data types provided by Clover, we had to modify the ISS of the existing `riscv-vp` to make use of these data types and thus execute RV32 instructions based on concolic operands. For this purpose, we had to switch the underlying data type, stored by the register file and the memory, from concrete integers to concolic values (as provided

[4]https://github.com/agra-uni-bremen/clover

```
1  case Opcode::BEQ: {
2  //----------[ CONCRETE BEQ IMPL. ]----------
3    if (regs[RS1] == regs[RS2]) {
4      pc = last_pc + instr.B_imm();
5      trap_check_pc_alignment();
6    }
7    break;
8  //----------[ CONCOLIC BEQ IMPL. ]----------
9    auto compare = regs[RS1]->eq(regs[RS2]);
10   auto is_true = eval(res->concrete);
11   if (is_true) {
12     pc = last_pc + instr.B_imm();
13     trap_check_pc_alignment();
14   }
15
16   track_and_trace_branch(is_true, compare);
17   break;
18 //-----------------[ END ]-----------------
19 }
```

Listing 3.1.: Concrete and concolic implementation of `BEQ` instruction.

by Clover). Furthermore, we had to modify each RISC-V instruction implementation (provided by `riscv-vp`) to operate on these concolic values by utilizing the data types supplied by Clover. For branch instructions, we additionally implemented tracking of encountered branch conditions as a prerequisite for our DSE (Section 2.3) algorithm implementation.

For illustrative purposes, a comparison of the concrete and concolic RISC-V `BEQ` branch instruction implementation is provided in Listing 3.1. The instruction compares two register operands for equality and jumps to a specified address (relative to the program counter) if the two register operands are equal. The original concrete implementation of this instruction is shown in Line 3 - Line 7. This implementation compares two 32-bit integers using the `==` operator (Line 3) and if they are equal, it modifies the program counter accordingly (Line 4) and checks if it is still aligned correctly (Line 5). The concolic implementation of this instruction is shown in Line 9 - Line 17. Compared to the concrete implementation, it operates on concolic—instead of concrete—values. In order to compare the register operands, it constructs a new equality expression (Line 9) and evaluates the concrete part of this expression in Line 10. If the concolic register operands are equal, it performs the same jump as the concrete implementation (Line 11 - Line 14). However, contrary to the concrete implementation, it always tracks the result of the evaluation and the equality expression itself via `track_and_trace_branch` in Line 16. This allows iteratively creating an execution tree, similar to the one visualized in Figure 2.3. When execution terminates, a DSE algorithm is employed to select a random branch condition from this tree and negates it using an SMT solver, thereby determining a new assignment for concolic input values (see Section 2.3).

SymEx-VP supports `RV32IMAC_zicsr`, i.e. the 32-bit base instructions [153, Chapter 2], the compressed instructions [153, Chapter 16], the atomics instructions [153, Chapter 8], the multiplication instructions [153, Chapter 7], and the Control and Status Register (CSR) instructions [153, Chapter 9] of the RISC-V specification. Notably, SymEx-VP therefore supports the same extensions as the SiFive HiFive 1.

3.1.2.4. SystemC Integration

Apart from the SystemC-independent Clover library, our architecture also contains SystemC-specific components. Most importantly, the TLM extension for transporting concolic values over the TLM bus. We have implemented our extension for concolic values as an *ignorable extension*, i.e. “any component other than the component that added the extension is permitted to behave as if the extension were absent” [180, Section 14.21.1.1]. Therefore, existing SystemC TLM components, which receive a TLM transaction with our extension attached, will continue to work without any modifications. Nonetheless, our extension enables the injection of concolic test inputs by modifying existing peripherals to return concolic values on MMIO operations abstracted via TLM transactions (an example will be provided in Subsection 3.1.3).

As depicted in Figure 3.1, the executed firmware is then explored based on injected concolic values. For this purpose, the ISS collects branches which depend on these values. Upon simulation termination, the DSE implementation in Clover is used to negate a random branch condition and generate new input values. This requires restarting the SystemC simulation, which is not natively supported by SystemC version 2.3.3. Therefore, we have modified SystemC to include support for in-place simulation restarting and proposed our modifications to SystemC developers. A refined version of these modifications is currently in the process of being integrated by SystemC developers.[5]

[5] `https://github.com/accellera-official/systemc/issues/8`

3.1.2.5. SymbolicCTRL

In order for the SystemC simulation to be restarted with new input values, the executed firmware needs to signal the end of the current execution to the VP. Since firmware does not terminate, this requires the verification engineer to select an appropriate termination point as part of the test setup. This termination point is specified by having the firmware communicate with the VP via a MMIO register provided by a special `SymbolicCTRL` peripheral. Upon receiving a write transaction for this register, the VP will terminate execution and restart the SystemC simulation with new input values. Similarly, the `SymbolicCTRL` peripheral also provides a dedicated MMIO register to allow the firmware to signal an error condition. This allows classifying enumerated execution paths in the VP. For example, the firmware may signal an error condition upon invocation of a panic handler or an assertion failure. A detailed discussion of error detection techniques is provided in Chapter 5.

Lastly, the `SymbolicCTRL` peripheral also allows injecting a fixed amount of unconstrained symbolic bytes into the firmware execution via MMIO. This enables a verification engineer to write custom test harnesses that tests individual functions (i.e. units) of a firmware by passing symbolic bytes, retrieved from the `SymbolicCTRL` peripheral, directly to these functions (see Subsection 3.1.5).

3.1.3. Peripheral Modeling Example

In this section, we will demonstrate how peripherals are modeled in SymEx-VP using SystemC TLM and how the aforementioned TLM extension for concolic values is used to inject concolic inputs through peripherals. For this purpose, we will present a simplified SystemC model of the SiFive Universal Asynchronous Receiver-Transmitter (UART) peripheral as used by the HiFive1 [171, Chapter 17]. We chose this peripheral because it is comparatively simple and thus well suited for illustrative purposes. Furthermore, it is widely supported by existing embedded operating systems. For this reason, a concolic UART peripheral allows testing the input handling routines of these systems, including the network stack via a Serial Line Internet Protocol (SLIP) [155] network interface. More information on SLIP and network stack testing will be provided in Subsection 3.1.4.

3.1.3.1. SiFive UART

The SiFive UART is specified in the SiFive FE310-G000 Manual [171]; it has two 8-bit transmit and receive FIFOs with a configurable watermark for interrupts. Advanced fea-

Table 3.1.: Memory map of the UART on the SiFive HiFive1 [171, Table 49].

Offset	Name	Description
`0x00`	`txdata`	Transmit data register
`0x04`	`rxdata`	Receive data register
`0x08`	`txctrl`	Transmit control register
`0x0C`	`rxctrl`	Receive control register
`0x10`	`ie`	UART interrupt enable
`0x14`	`ip`	UART interrupt pending
`0x18`	`div`	Baud rate divisor

tures, such as hardware flow control or parity bits, are not supported. The memory map of the SiFive UART is shown in Table 3.1, it consists of seven 32-bit memory-mapped registers. The first two (`txdata` and `rxdata`) are used to interact with the transmit/receive FIFOs, the watermark for these is configured via the `txctrl` and `rxctrl` registers. The UART uses a single interrupt to signal ready-to-receive/ready-to-transmit. Interrupts are enabled via the `ie` register. The `ip` register is used to distinguish between ready-to-transmit/ready-to-receive in a UART-specific interrupt handler. The remaining `div` register allows configuring the baud rate [171, Section 17.9].

In order to model memory-mapped registers, loosely-timed[6] SystemC peripheral models will register a callback function, which is invoked for each transaction received for this peripheral via the TLM bus [180, Section 11.1.1]. An exemplary implementation of this function is shown in Listing 3.2. The function takes two parameters: a generic TLM payload, which represents the transaction that should be performed (Line 1), and a delay for annotating timing information (Line 2). The TLM generic payload is a C++ class that includes the memory address which should be accessed in the peripheral and information on how this access should be performed (e.g. whether the register should be read or written). Based on the targeted memory address, the code in Listing 3.2

[6]Loosely-timed and approximately-timed are different SystemC coding styles [180, Section 10.3]. We focus on the former since accurate timing analysis is not the focus of this section.

performs a case distinction for each memory-mapped register supported by the SiFive UART (Line 6 - Line 30). Depending on the TLM transaction command, the internal value for this register is then either modified (TLM write command) or its current value is returned (TLM read command). The excerpt in Listing 3.2 focuses on the latter TLM command and presents the handling for the `rxdata` and `ip` register. Other registers have been omitted since they are not used to pass environmental input to the firmware. In Listing 3.2, both the transmit and receive FIFOs of the SiFive UART are implemented using separate queues (`tx_fifo` and `rx_fifo`). The oldest element from the receive queue can be retrieved by reading the `rxdata` register (Line 7 - Line 18). If the receive queue is empty, the Most Significant Bit (MSB) is set in the register to indicate this condition (Line 9). This is possible since the register is 32-bit wide, but only 8-bit are used for receiving data from the underlying FIFO. The implementation for the memory-mapped `ip` register checks if either the receive or the transmit queue exceeds the configured watermark and if so, sets the pending interrupt register bits accordingly (Line 19 - Line 25). Lastly, the address—given in the TLM transaction—is converted to a pointer, which points to one of the internal 32-bit variables representing the different memory-mapped registers (Line 32). The data stored in this internal variable is then copied to the data pointer of the TLM transaction (Line 33 - Line 34), thereby allowing the initiator of the TLM transaction to access the value of the targeted memory-mapped register.

3.1.3.2. Injecting Concolic Values

In accordance with prior work, we believe the majority of errors to occur in input handling routines of embedded IoT firmware [159]. In the SiFive UART context, we therefore focus on the `rxdata` register as this is the register used to pass (potentially) untrusted input from the environment to the firmware. As such, the firmware must be able to handle arbitrary data received from this memory-mapped register. Other memory-mapped registers, provided by the SiFive UART, are not directly controlled by the environment and thus do not inherit this requirement.

In order to explore all program states reachable through environmental input via the `rxdata` register, we inject concolic values into the firmware execution upon reading this register. This is achieved by utilizing the TLM extension for concolic values (Subsection 3.1.2) in our SystemC peripheral model for the SiFive UART. Using this extension, we refine the handling of the TLM read command for this register. For this purpose, we change the underlying data type for the receive queue (`rx_fifo`) from a `uint8_t` to a

```cpp
void transport(tlm::tlm_generic_payload &trans,
               sc_core::sc_time &delay) {
  auto cmd = trans.get_command();
  auto addr = trans.get_address();
  if (cmd == tlm::TLM_READ_COMMAND) {
    switch (addr) {
    case RXDATA_REG_ADDR:
      if (rx_fifo.empty()) {
        this->rxdata = 1 << 31; // set MSB
      } else {
        uint8_t next_byte;
        next_byte = rx_fifo.front();
        rx_fifo.pop();

        // Zero-extend to 32-bit (MSB unset)
        rxdata = (uint32_t)next_byte;
      }
      break;
    case IP_REG_ADDR:
      ip = 0; // reset IP register value
      if (tx_fifo.size() < WATERMARK(txctrl))
        ip |= UART_TXWM; // ready-to-transmit
      if (rx_fifo.size() > WATERMARK(rxctrl))
        ip |= UART_RXWM; // ready-to-receive
      break;
    case TXDATA_REG_ADDR:
    case TXCTRL_REG_ADDR:
    case RXCTRL_REG_ADDR:
    // ...
    }

    auto reg = addr2pointer(addr);
    auto ptr = trans.get_data_ptr();
    memcpy(ptr, reg, sizeof(uint32_t));
  }
  // ...
}
```

Listing 3.2.: Blocking transport for a SiFive UART SystemC peripheral.

```
1  case RXDATA_REG_ADDR:
2    // Check if RX interrupt is enabled since many
3    // UART drivers drain rxdata before first use.
4    if (rx_fifo.empty() || !(ie & UART_RXWM)) {
5      rxdata = 1 << 31; // set MSB
6    } else {
7      std::shared_ptr<ConcolicValue> next_byte;
8      next_byte = rx_fifo.front();
9      rx_fifo.pop();
10
11     // Zero-extend to 32-bit (MSB unset)
12     auto reg = symbolic_byte->zext(32);
13
14     auto ext = new SymbolicExtension(reg);
15     trans.set_extension(ext);
16     rxdata = solver.getValue(reg->concrete);
17   }
```

Listing 3.3.: Inject concolic values on `rxdata` read using TLM extension.

`ConcolicValue`. In the SystemC module constructor for the peripheral, the receive queue is then initialized with a fixed amount of unconstrained symbolic values. By retaining the `rx_fifo` but filling it with values of a different type, we can refrain from modifying the handling of other registers (e.g. the `ip` register) which use the aforementioned queue. As such, only the implementation of the `rxdata` register needs to be adjusted to operate on concolic values. The resulting code is shown in Listing 3.3. The code in Listing 3.3 would be used to replace Line 7 - Line 18 in Listing 3.2. Similar to Listing 3.2, the implementation in Listing 3.3 takes the oldest element from the queue and zero-extends it to 32-bit to ensure that the MSB is unset (Line 11 - Line 12). In Listing 3.3, zero-extension is achieved through an operation on the concolic value (not through a type cast). Lastly, a C++ object for the TLM extension for concolic values is created in Line 14 and added to the TLM transaction object in Line 15. As per Subsection 3.1.2, the TLM extension is ignorable, thus the internal `rxdata` value also needs to be updated (Line 16) to ensure that the initiator of the TLM transaction can also choose to ignore the extension.[7]

Instead of injecting a fixed number of symbolic bytes, via the data bits of the `rxdata`

[7]The concrete value, stored in the `rxdata` register, would be copied to the data pointer of the TLM transaction as done in Line 34 of Listing 3.2.

register, it is also possible to declare the entire 32-bit `rxdata` register as symbolic. While this would allow reasoning about variable-size inputs (since the MSB empty bit would be part of the symbolic expression), our experiments indicate that doing so heavily increases the state space which ultimately leads to the state explosion problem (Section 2.3). The majority of input handling routines which receive data from a UART (e.g. an IPv6 [56] implementation receiving data via SLIP [155]) also mandate a minimum input size and thus discard all inputs below this size early, without performing any interesting input processing. For this reason, we deem states reached through inputs below a certain size to be negligible. We further concern ourselves with input generation heuristics and state space explosion in Chapter 6.

3.1.4. Usage Scenario

In the following, we will illustrate how firmware for constrained IoT devices can be tested using SymEx-VP. For this purpose, we utilize the concolic SiFive UART peripheral described in Subsection 3.1.3 to test the network stack of the RIOT [9] operating system via SLIP [155].

3.1.4.1. Setup

SLIP is a network protocol standardized by the Internet Engineering Task Force (IETF) which allows transmitting Internet Protocol (IP) datagrams over a serial line interface, e.g. as provided by the SiFive UART. For this purpose, SLIP defines a framing format which allows determining the start/end of an IP datagram within a continuous stream of bytes. The framing format consists of two control characters: a character which identifies the end of a frame and a character to escape the former control character within the primary data of an IP datagram [155, p. 2]. In order to transmit concolic SLIP frames over our SiFive UART, we terminate our fixed-size concolic input with the SLIP end frame control character. Furthermore, we need to ensure that this control character is always escaped within the primary data of the concolic IP datagram.[8] Unfortunately, expressing the SLIP escaping rules as a symbolic expression for each symbolic byte leads to complex SMT queries. We mitigate this problem by constraining the symbolic bytes within a frame to never match the end frame control character.[9]

[8]Otherwise, the concolic testing engine would generate inputs below our desired minimum size.

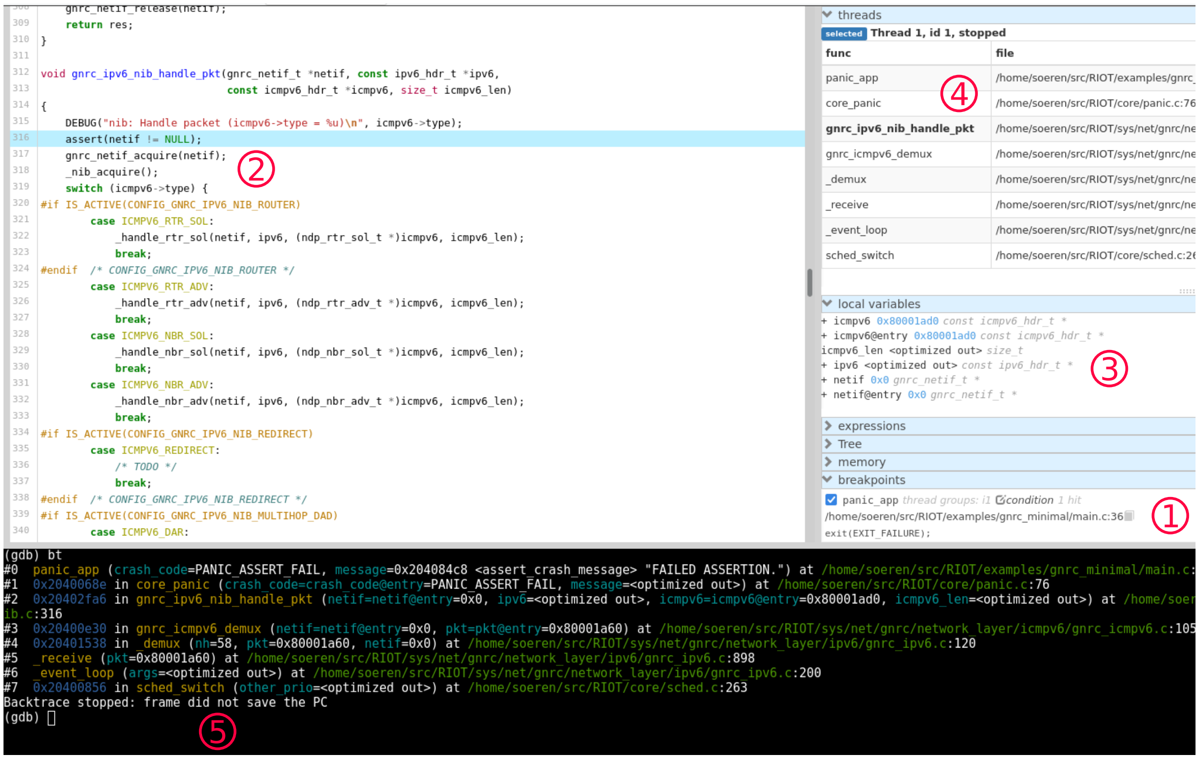
[9]Our setup is thus unable to generate truly arbitrary inputs. This limitation can be best addressed by utilizing a different network peripheral that does not rely on input escaping (e.g. Ethernet).

Based on this SLIP-aware variant of the concolic SiFive UART, we can test the network stack of different IoT operating systems. To illustrate our bug finding workflow with SymEx-VP, the following subsections describe the discovery of a globally reachable failing assertion we found in RIOT's network stack. The default network stack of the RIOT operating system is called GNRC [9, Section 10], it is further described in a publication by Lenders et al. [112]. Since RIOT provides a modular operating system architecture, each component of the RIOT network stack is implemented in a separate module and runs in its own operating system thread [112, Section 4]. In order to mitigate the state explosion problem, it makes sense to employ a divide-and-conquer strategy and test these modules separately. However, instead of writing a custom test harness for each module, we use the existing `examples/gnrc_minimal` application provided by RIOT and target individual network protocol implementations by constraining our symbolic input accordingly.

3.1.4.2. Testing & Debugging

We focus on testing the ICMPv6 [76] implementation, as used by the `gnrc_minimal` example application. In order to do so, the upper layers of the input network packet are treated as concrete, thereby having SymEx-VP always explore paths within the ICMPv6 implementation. Furthermore, the application is configured to use a SLIP network interface, based on our SiFive UART model, and compiled for the SiFive HiFive1. The resulting Executable and Linkable Format (ELF) binary is then executed with SymEx-VP's `hifive-vp` executable, which models the SiFive HiFive1 and aims to be binary-compatible with this platform. The executed RIOT firmware image will read concolic input bytes from the `rxdata` register of the SLIP-aware SiFive UART. These bytes are then passed to the GNRC network stack and processed as an IPv6 [56] datagram with an encapsulated ICMPv6 payload. After processing a single packet, execution is terminated. Lastly, the SymEx-VP exploration engine will restart the program with new inputs until either all paths have been explored or an error is encountered. Since the upper layers of the network packet are partially concrete, exploration will focus on RIOT's ICMPv6 implementation (`gnrc_icmpv6`). After executing 75 paths in 52 seconds, a kernel panic in RIOT is encountered, thus SymEx-VP aborts further exploration.[10] The kernel panic is caused by a failing assertion; however, without any further debugging, the reason for the assertion failure is unclear.

[10] SymEx-VP detects this kernel panic through a modified panic handler which signals an error condition to the `SymbolicCTRL` peripheral.



```c
    gnrc_netif_release(netif);
    return res;
}

void gnrc_ipv6_nib_handle_pkt(gnrc_netif_t *netif, const ipv6_hdr_t *ipv6,
                              const icmpv6_hdr_t *icmpv6, size_t icmpv6_len)
{
    DEBUG("nib: Handle packet (icmpv6->type = %u)\n", icmpv6->type);
    assert(netif != NULL);
    gnrc_netif_acquire(netif);
    _nib_acquire();
    switch (icmpv6->type) {
#if IS_ACTIVE(CONFIG_GNRC_IPV6_NIB_ROUTER)
        case ICMPV6_RTR_SOL:
            _handle_rtr_sol(netif, ipv6, (ndp_rtr_sol_t *)icmpv6, icmpv6_len);
            break;
#endif  /* CONFIG_GNRC_IPV6_NIB_ROUTER */
        case ICMPV6_RTR_ADV:
            _handle_rtr_adv(netif, ipv6, (ndp_rtr_adv_t *)icmpv6, icmpv6_len);
            break;
        case ICMPV6_NBR_SOL:
            _handle_nbr_sol(netif, ipv6, (ndp_nbr_sol_t *)icmpv6, icmpv6_len);
            break;
        case ICMPV6_NBR_ADV:
            _handle_nbr_adv(netif, ipv6, (ndp_nbr_adv_t *)icmpv6, icmpv6_len);
            break;
#if IS_ACTIVE(CONFIG_GNRC_IPV6_NIB_REDIRECT)
        case ICMPV6_REDIRECT:
            /* TODO */
            break;
#endif  /* CONFIG_GNRC_IPV6_NIB_REDIRECT */
#if IS_ACTIVE(CONFIG_GNRC_IPV6_NIB_MULTIHOP_DAD)
        case ICMPV6_DAR:
```

threads

selected Thread 1, id 1, stopped

func	file
panic_app	/home/soeren/src/RIOT/examples/gnrc_
core_panic	/home/soeren/src/RIOT/core/panic.c:76
gnrc_ipv6_nib_handle_pkt	/home/soeren/src/RIOT/sys/net/gnrc/ne
gnrc_icmpv6_demux	/home/soeren/src/RIOT/sys/net/gnrc/ne
_demux	/home/soeren/src/RIOT/sys/net/gnrc/ne
_receive	/home/soeren/src/RIOT/sys/net/gnrc/ne
_event_loop	/home/soeren/src/RIOT/sys/net/gnrc/ne
sched_switch	/home/soeren/src/RIOT/core/sched.c:26

local variables

```
+ icmpv6 0x80001ad0 const icmpv6_hdr_t *
+ icmpv6@entry 0x80001ad0 const icmpv6_hdr_t *
icmpv6_len <optimized out> size_t
+ ipv6 <optimized out> const ipv6_hdr_t *
+ netif 0x0 gnrc_netif_t *
+ netif@entry 0x0 gnrc_netif_t *
```

expressions
Tree
memory
breakpoints

✓ panic_app *thread groups: i1* *condition* *1 hit*
/home/soeren/src/RIOT/examples/gnrc_minimal/main.c:36
exit(EXIT_FAILURE);

```
(gdb) bt
#0  panic_app (crash_code=PANIC_ASSERT_FAIL, message=0x204084c8 <assert_crash_message> "FAILED ASSERTION.") at /home/soeren/src/RIOT/examples/gnrc_minimal/main.c:
#1  0x2040068e in core_panic (crash_code=crash_code@entry=PANIC_ASSERT_FAIL, message=<optimized out>) at /home/soeren/src/RIOT/core/panic.c:76
#2  0x20402fa6 in gnrc_ipv6_nib_handle_pkt (netif=netif@entry=0x0, ipv6=<optimized out>, icmpv6=icmpv6@entry=0x80001ad0, icmpv6_len=<optimized out>) at /home/soer
ib.c:316
#3  0x20400e30 in gnrc_icmpv6_demux (netif=netif@entry=0x0, pkt=pkt@entry=0x80001a60) at /home/soeren/src/RIOT/sys/net/gnrc/network_layer/icmpv6/gnrc_icmpv6.c:105
#4  0x20401538 in _demux (nh=58, pkt=0x80001a60, netif=0x0) at /home/soeren/src/RIOT/sys/net/gnrc/network_layer/ipv6/gnrc_ipv6.c:120
#5  _receive (pkt=0x80001a60) at /home/soeren/src/RIOT/sys/net/gnrc/network_layer/ipv6/gnrc_ipv6.c:898
#6  _event_loop (args=<optimized out>) at /home/soeren/src/RIOT/sys/net/gnrc/network_layer/ipv6/gnrc_ipv6.c:200
#7  0x20400856 in sched_switch (other_prio=<optimized out>) at /home/soeren/src/RIOT/core/sched.c:263
Backtrace stopped: frame did not save the PC
(gdb) 
```

Figure 3.2.: Screenshot of a debugging session with SYMEX-VP's GDB stub.

Similar to other symbolic execution engines (e.g. KLEE [32]), SYMEX-VP is capable of generating a test case for each encountered error [32, Section 4.3]. In SYMEX-VP, these test cases are plain-text files which assign each symbolic variable (as identified by a unique variable name) a concrete value. Based on these test cases, the encountered error (i.e. the failing assertion in RIOT) can be debugged further. For this purpose, SYMEX-VP supports a replay mode which allows replaying generated test cases in the VP environment. In replay mode, SYMEX-VP performs concrete execution and the generated test cases are used as environmental inputs by the modeled peripherals.

During concrete execution in replay mode, it is possible to attach a debugger to SYMEX-VP via a provided GDB[11] stub. In Figure 3.2 a debugging session with a graphical GDB frontend[12] for the aforementioned kernel panic in RIOT is illustrated. The screenshot is divided into three sections: the source code (top left), general debugging information (top right), and the GDB output (bottom). As illustrated at position ①, the debugging session was created by configuring a breakpoint for the panic handler and

[11]https://www.gnu.org/software/gdb/
[12]https://gdbgui.com

executing the firmware concretely until it hits this breakpoint. At position ②, the source code for the assert statement (which causes the invocation of the panic handler) is shown. The assertion occurs in Line 316 of `ipv6/nib/nib.c` and is part of the function `gnrc_ipv6_nib_handle_pkt`. The source file is part of RIOT's Neighbor Discovery Protocol (NDP) [172] implementation for IPv6. NDP is based on ICMPv6 and is thus explored by the generated test inputs. Since the assertion at position ② is failing, the `netif` (network interface) pointer must be `NULL`. This can be confirmed by inspecting the local variable values for the currently selected function at position ③. Since the `netif` pointer is passed as a function parameter to `gnrc_ipv6_nib_handle_pkt` (Line 312), it must already be `NULL` in the caller. At position ④, the backtrace which leads to the assertion failure is shown. The currently selected function in the backtrace is marked with a bold font. From the backtrace we can deduce that `gnrc_icmpv6_demux` is responsible for passing the `netif` pointer to `gnrc_ipv6_nib_handle_pkt`. By inspecting the function parameters of each executed function at position ⑤ we can conclude that the `NULL` pointer first appears as a function parameter in the `_demux` function of the `gnrc_ipv6` module and must thus originate in the `_receive` function. The `_receive` function attempts to extract information about the network interface based on metadata which must be set by the utilized network interface module. Unfortunately, RIOT's implementation of the SLIP network interface did not set this metadata and thus caused the assertion failure in the NDP implementation. The outlined assertion failure was previously unknown; it was fixed by RIOT developers in Git commit 3384c32 by adjusting the SLIP network interface implementation to correctly configure this metadata information.[13]

Since this bug originates in the low-level SLIP implementation, which receives data directly from the utilized SiFive UART, this bug illustrates the importance of testing IoT firmware directly through peripheral inputs instead of operating on a higher abstraction level. Furthermore, the preceding discussion demonstrates that SymEx-VP is capable of finding deep bugs in real-world firmware for constrained devices. The illustrated bug occurs after passing concolic input bytes from the UART to the SLIP network interface all the way through the IPv6 and ICMPv6 implementations and ultimately ends up causing an error in the NDP implementation. The complexity of the discussed bug also shows the benefits of the debugging facilities provided by SymEx-VP. Due to support for test case generation, test case replaying, and GDB-based debugging, it is possible to easily debug issues found through concolic testing.

[13]`https://github.com/RIOT-OS/RIOT/commit/3384c327a7c82afde033033f00123ad9700aae7d`

3.1.5. Evaluation

In the following, we focus on the quantitative dimension by providing evidence that SymEx-VP can be used not only with RIOT but also with other operating systems and Software Development Kits (SDKs) for the embedded domain. We believe this property is important to ensure that SymEx-VP integrates well with the heterogeneous ecosystem described in Section 1.1. For this purpose, we employ SymEx-VP for testing a variety of different firmware images with minimally invasive changes for integration purposes. This is achieved by injecting concolic values via the MMIO peripheral interface using our provided SystemC TLM extension. In order to illustrate the advantages of this approach, we compare it to concolic unit tests of individual firmware units (i.e. functions) [162]. For this purpose, we use a manually created test harness where concolic values originate in the harness using the `SymbolicCTRL` peripheral (see Subsection 3.1.2). The evaluation is designed to be reproducible and artifacts are available on Code Ocean [199].

In total, we tested ten different components of four embedded operating systems and SDKs using SymEx-VP. We focus specifically on IoT operating systems since, in accordance with Chapter 1, we believe these to be the most complex kind of embedded firmware images. Regarding IoT operating systems, we focused primarily on RIOT[14] and Zephyr[15] since we believe these two to be the most common operating systems with RISC-V support in the IoT domain. Furthermore, we performed tests with the more complex Apache NuttX[16] operating system and the `zig-riscv-embedded`[17] bare-metal Constrained Application Protocol (CoAP) [167] firmware which uses a HiFive1 SDK for the Zig[18] programming language. All tests have been performed using the `hifive-vp` model, provided by SymEx-VP, which targets the SiFive HiFive1 platform.

We tested different components of these operating systems by executing them for 30 min with SymEx-VP. The results of the performed tests are shown in Table 3.2. For each tested operating system, we list the test target (i.e. the tested component), the test type, the amount of code modified for the test setup, the time spent in the SMT solver, the amount of execution paths found, and the number of paths found per second. Regarding the test type, we distinguish between the aforementioned tests of individual component units (UNIT) and tests of entire existing applications where

[14]`https://riot-os.org`
[15]`https://zephyrproject.org/`
[16]`https://nuttx.apache.org/`
[17]`https://github.com/nmeum/zig-riscv-embedded`
[18]`https://ziglang.org`

Table 3.2.: Execution statistics for applications from different IoT operating systems.

					Execution Paths	
OS	Target	Test Type	LOC modified	Solver Time	#Paths	Paths/s
Riot	`default`	UART	46	19 min	6275	3.5
	`gnrc_minimal`	SLIP	67	25 min	1241	0.7
	`sock_dns`	UNIT	198	22 min	7019	3.9
	`jsmn`	UNIT	135	13 min	15954	8.9
	`ipv6_addr`	UNIT	132	10 min	14541	8.1
Zephyr	`shell`	UART	52	6 min	7159	4.0
	`ipaddr`	UNIT	95	17 min	7029	3.9
	`json`	UNIT	129	8 min	16874	9.4
NuttX	`shell`	UART	43	2 min	1357	0.7
Zig	`zig-hifive1`	UART	34	6 min	8822	4.9

values are injected through the UART/SLIP peripheral introduced in Subsection 3.1.3 and Subsection 3.1.4. As shown in Table 3.2, UNIT tests have a higher integration effort (as measured in LOC). For UART/SLIP-based tests types, we only need to modify the existing firmware image to signal the end of current execution and invocations of the panic handler to SymEx-VP as described in Subsection 3.1.2.5. For UNIT tests, it is additionally necessary to write a test harness for selected units of the tested component since concolic values do not originate automatically in the MMIO peripheral interface for these tests. For some test targets, the desired units are not accessible directly and require mocking certain data structures, thereby increasing the integration effort. This is, for example, the case for the `sock_dns` RIOT component which receives input via a UDP [146] socket data structure that had to be mocked and modified to return concolic values.

Depending on the test target, a side effect of injecting concolic values via the MMIO peripheral interfaces is an increase in solver time. In Table 3.2, the target where the most time was spent in the SMT solver was the `gnrc_minimal` RIOT target which is an existing example application providing a minimal configuration of RIOT's GNRC network stack. For this application, concolic values originate in the SLIP peripheral and are passed through the entire network stack. Compared to testing of individual units, this enforces more complicated constraints on these values and thus results in increased solver time. Nonetheless, this demonstrates that our approach is capable of also executing these complex firmware images. In general, the bottleneck in terms of performance (as measured by execution paths per second) is primarily the SMT solver, not the performance penalty caused by the detailed reasoning about low-level hardware details enabled by SystemC. The two Zephyr targets (`shell` and `json`) as well as the Apache NuttX operating system and the `zig-hifive1` firmware image differ in this regard. For these targets, more time is spent with concrete execution than with solving constraints. This is likely due to the fact that we currently restart the SystemC simulation for each new generated input and these targets have more complex boot code compared to RIOT. This boot code has to be re-executed for each execution path. In future work, this could be mitigated through the use of simulation snapshots, i.e. by only executing the boot code once.

In summary, our experiments illustrate that SymEx-VP is capable of executing a wide range of different firmware images from low-end IoT firmware (such as RIOT) to complex real-time operating systems (such as Apache NuttX). By injecting concolic values through the MMIO peripheral interface, we can test new applications with minimal integration effort. Nonetheless, it is also possible to test individual firmware units with SymEx-VP, thereby trading integration effort for performance (as measured in discovered execution paths per second). By operating on the RISC-V machine code level, SymEx-VP is also capable of executing code written in different programming languages. While the majority of tested firmware images are written in C, our experiments also demonstrate our ability to execute code written in other languages (e.g. the Zig programming language). We believe this to be important since safer alternatives to C/C++ (such as Zig and Rust) are on the rise in the embedded domain [114].

3.1.6. Related Work

An overview of prior work on employing dynamic verification techniques for testing embedded firmware images is provided by Fasano et al. [66]. According to this publication, prior work has already experimented with emulation-based firmware testing, for example by utilizing QEMU [16]. Unfortunately, QEMU does not provide a modeling language for hardware peripherals (e.g. SystemC) and does thus not support reasoning about low-level behavior of a wide range of peripherals. In order to address this deficiency, existing publications propose hybrid emulation which combines execution in QEMU with execution on real hardware, thereby enabling execution of code interacting with hardware peripherals [210, 127, 109]. A popular approach in this regard is Avatar^2 [127], which is the successor to Avatar [210], and forwards program state to the real hardware using GDB. A disadvantage of hybrid emulation is the performance penalty induced by moving program state between an emulator and the real hardware. Prior work by Koscher et al. attempts to improve the performance of this technique [109]. However, it also remains challenging to propagate concolic or symbolic values across the emulator/device boundary.

A different direction pursued in prior work is executing LLVM Intermediate Representation (IR) to achieve symbolic execution of firmware images using the KLEE [32] symbolic execution engine [49, 54]. In this regard, Inception [49] also utilizes hybrid emulation while Fie [54] approximates peripheral behavior through peripheral models. Peripheral approximation (also referred to as *rehosting*) is also pursued in other publications to address challenges regarding utilized peripherals [77, 35, 68, 122, 66]. Feng et al. present P^{2}IM, which automatically generates peripheral models during execution [68]. Mera et al. extend P^{2}IM with support for Direct Memory Access (DMA) [122]. A different approximation approach is taken by Cao et al. which return concolic values to approximate peripheral inputs [35]. All of these approaches do not use accurate peripheral models and are therefore not sound, i.e. test results can include false-positives.

While the publications discussed so far are largely independent of the utilized operating system or SDK, prior work has also experimented with operating on a higher abstraction level to deal with peripheral interactions. An example of this approach is HALucinator which operates on a HAL and thus does not need to model or approximate low-level peripheral behavior [157]. Unfortunately, this approach is not agnostic, i.e. only applicable to firmware images using a certain HAL.

3.1.7. Discussion and Future Work

In this section, we have primarily focused on environmental modeling via SystemC in a dynamic testing context. Regarding future work, the concolic testing engine provided by Clover presently only implements basic concolic testing techniques. For example, as a memory model we currently employ address concretization, which means that symbolic values are concretized when they are used as memory addresses, thus potentially causing some paths to be missed [10, Section 3.2]. Furthermore, we employ a path selection algorithm which selects random nodes from the execution tree but prefers upper nodes in the tree and does not use advanced search heuristics. We plan to improve the underlying concolic testing engine, provided by Clover, in future work.

Additionally, we have not yet optimized SymEx-VP for performance. As illustrated in Subsection 3.1.5, we already achieve decent execution performance for RIOT and Zephyr but in more complex firmware images (e.g. the one provided by Apache NuttX) fewer execution paths are discovered per second. The major limitation in this regard is that we restart the SystemC simulation for each new input, thereby re-executing the boot code every time. In future work, we plan to investigate whether it is feasible to store different SystemC simulation states instead of restarting the simulation every time.

3.1.8. Conclusion

We presented SymEx-VP, an open source VP-based symbolic execution engine specifically tailored to challenges regarding firmware testing (Section 1.1) and the first to support accurate peripheral models by utilizing a hardware modeling language (SystemC). SymEx-VP uses concolic testing to explore different paths through firmware images and injects concolic values into the executed firmware through the MMIO peripheral interface (Subsection 3.1.2). Peripherals are modeled in SymEx-VP using the existing SystemC TLM hardware modeling language. SystemC TLM models peripherals based on a bus abstraction; by utilizing a TLM extension mechanism, we can transport concolic values over this bus. We illustrated that this TLM extension can be used in conjunction with existing SystemC peripheral models by presenting an example integration for the SiFive UART (Subsection 3.1.3). Based on this model, we have demonstrated that components of existing firmware images can be tested and debugged using SymEx-VP (Subsection 3.1.4). We have also provided evidence that our TLM extension mechanism allows us to employ SymEx-VP—with minimal integration effort—for testing a variety of different operating systems in the heterogeneous embedded and IoT ecosystems (Subsection 3.1.5). The presented SymEx-VP is fully open source and thereby serves as a foundation for enabling research on advanced symbolic execution techniques for the embedded domain.

3.2. Injecting Symbolic Values using Peripheral Overlays

In Section 3.1, we have presented an integration of SystemC-based VPs with symbolic execution for the purpose of automated firmware testing. This integration allows us to model peripheral behavior on a high abstraction level through SystemC TLM, thereby addressing challenges regarding environment interactions described in Section 1.1. In order to integrate SystemC TLM hardware models with symbolic execution, we have proposed an extension mechanism for transporting symbolic values over the TLM bus. Using this extension mechanism, Subsection 3.1.3 presented a symbolic variant of the SiFive UART. The symbolic UART model allows injecting symbolic values into the SystemC simulation with minimal changes to the firmware image itself, thereby enabling tests of a wide range of different firmware images and achieving compatibility with the heterogeneous embedded ecosystem. Unfortunately, we had to modify an existing SystemC model of the SiFive UART to integrate it with symbolic execution. In an industrial context, modifications of existing peripheral models are challenging for the following reasons:

1. Existing SystemC models may be vendor-supplied; in this case, it can be difficult to sufficiently understand and modify the existing model to integrate it with symbolic execution. In the worst case, the source code may not even be available, rendering such modifications practically impossible.

2. Modifications of an existing model, especially if it is complex and vendor-supplied, may introduce bugs. Furthermore, if the model was formally verified, proofed properties may no longer hold after the modifications. This is crucial as incorrect hardware models may lead to bugs being missed in the tested firmware.

3. When employing virtual prototyping to develop new embedded systems, one would need to maintain two variants of a peripheral: the concrete, unmodified variant (for concrete simulation) and the symbolic variant (for symbolic simulation). This can be laborious and increases development time, thereby negatively affecting the time-to-market of an embedded system (see Section 2.1).

In order to overcome these limitations, we contribute an *overlay mechanism* that improves upon our prior work as described in Section 3.1. Overlays intercept TLM transactions to process the symbolic extension before and after the transaction is routed to the existing SystemC-based peripheral. This allows augmenting existing peripherals with our TLM extension without requiring modifications of these peripherals. Therefore, the overlay mechanism reduces the integration effort, as existing SystemC peripherals can be reused unmodified. A case study, conducted with an exemplary SystemC TLM model, demonstrates the advantages of this approach.

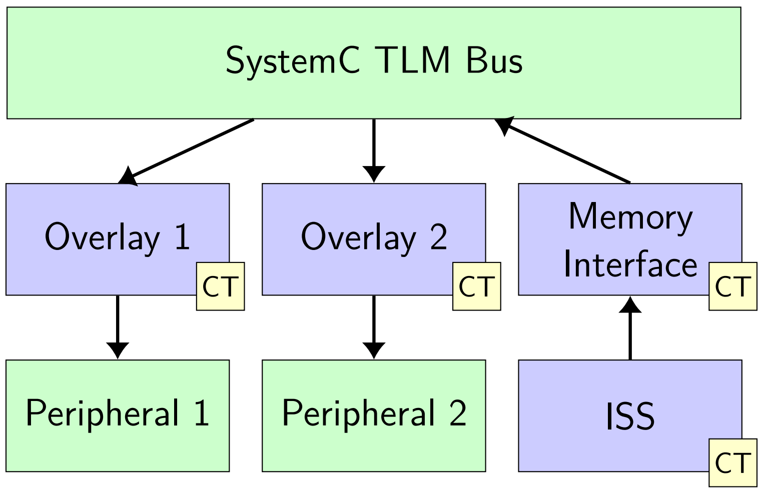


Figure 3.3.: Example integration of overlays with SystemC TLM.

3.2.1. SystemC TLM Overlays

SystemC TLM models hardware behavior at the transaction level based on a bus abstraction (see Section 2.1). Hardware peripherals, modeled in SystemC TLM, communicate by exchanging TLM transactions, which are transported over the bus. The central component for their transport is the TLM bus. Conceptually, the bus acts as a router; it is responsible for transporting transactions from an initiator (where the transaction originates) to a target (which processes the transaction) [21, Section 16.5]. Usually, multiple targets (such as UARTs, sensors, or memory) are attached to the same bus. In order to determine, to which target the transaction should be forwarded, the bus consults the transaction address encapsulated in the TLM payload. Essentially, the bus thereby acts as an address decoder which implements the memory map of the modeled embedded system [21, Section 16.6]. For example, if a sensor peripheral is mapped into memory from address `0x10000000` to `0x10001000` and a TLM transaction—received by the router—has an address within this range, then it is forwarded to this sensor peripheral.

In Section 3.1, we leveraged a TLM extension mechanism to transport symbolic values alongside TLM transactions. As we have seen in Subsection 3.1.3.2, this extension needs to be added to existing TLM payloads using the standardized `set_extension` method [180, Section 14.21]. While previously, this was achieved by modifying the existing TLM model this is by no means a necessity as we can also add the extension separately during forwarding of the transaction to the target. Instead of modifying the bus itself for this purpose, we inject a new component between the bus and the target peripheral. We refer to this new component as an overlay for the target peripheral. Figure 3.3 illustrates the use of overlays in a VP context. In this figure, two peripherals are provided. Firmware simulated by the VP's ISS interacts with these peripherals using MMIO through the memory interface. However, contrary to the standard SystemC architecture described previously, the peripherals are not attached directly to the TLM bus. Instead, a new overlay component is added between the peripherals and the bus. This component intercepts all TLM transactions and is responsible for adding the extension for symbolic values to outgoing transactions that are sent from the peripheral. The advantage of this approach is that the peripherals themselves do not need to be modified. This is illustrated in Figure 3.3 where components modified for an integration with concolic testing (CT) are colored blue while unmodified components are colored green. By leaving the existing peripheral models as-is, the proposed overlay mechanism resolves the challenges identified in Section 3.2.

3.2.2. Implementation

This section describes an integration of SymEx-VP, as presented in Section 3.1, with our proposed TLM overlay mechanism. Furthermore, an exemplary symbolic overlay for an existing SystemC model is discussed.

3.2.2.1. Integration with SymEx-VP

Figure 3.4 provides an overview of our integration of SystemC TLM overlays with SymEx-VP. The software architecture is largely unmodified, compared to the one presented in Subsection 3.1.2. This is a good thing as it illustrates that our proposed overlay mechanism can be integrated into existing VPs in a minimally invasive way. As described in Subsection 3.2.1, overlays are attached between the SymEx-VP TLM bus and the peripherals (e.g. sensors or UARTs) provided by SymEx-VP (see the bottom left corner of Figure 3.4). The peripherals themselves remain unmodified and interact with existing SymEx-VP components, e.g. the interrupt controller as before. As a result, this overlay mechanism reduces the effort required to integrate new SystemC peripherals with SymEx-VP. More specifically, as SymEx-VP is based on `riscv-vp` [85] from prior work, this also eases porting existing `riscv-vp` peripheral models for RISC-V hardware platforms to SymEx-VP. As such, the overlay mechanism improves the overall process for integrating existing VPs with symbolic execution. Therefore, it eases applying our proposed symbolic execution approach to additional hardware platforms and architectures. In the following section, we will illustrate an example peripheral integration. For

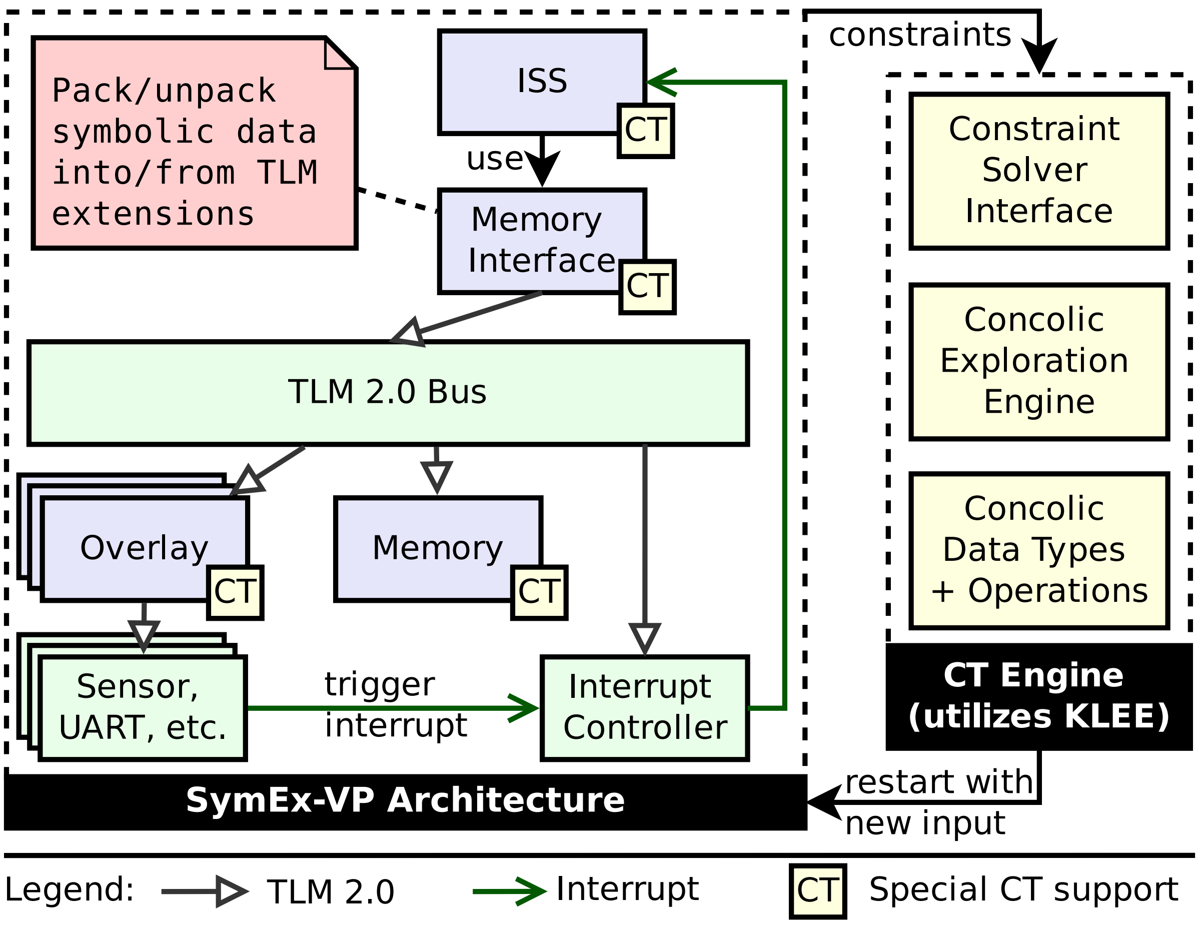


Figure 3.4.: Overview on our overlay mechanism integration with VPs.

this purpose, we propose a methodology where the symbolic overlay is iteratively refined in order to explore an exemplary embedded firmware.

3.2.2.2. Peripheral Overlay Example

As an example peripheral, we will again consider the SiFive UART, as used previously in Subsection 3.1.3. To summarize its functionality, it provides memory-mapped registers to receive and transmit data as well as configuration registers to enable interrupts and configure the watermark level (how many elements should be received before triggering an interrupt). The UART provides two small FIFOs to store received and process transmitted characters. In order to receive data, the `rxdata` register is read and returns a 32-bit value where the highest bit indicates if the FIFO is empty (the bit is set) and the lower 8-bit are the received character (if not empty).

Listing 3.4 shows a basic embedded software driver, that copies data from an RX

```
 1 void copy_driver() {
 2   //-------------------------[ VARIANT-1 ]---------
 3   char c = read_rx_uart();
 4   write_tx_uart(c);
 5   //-------------------------[ VARIANT-2 ]---------
 6   char c = read_rx_uart();
 7   assert (c != '#');  // assume sender filters '#'
 8   write_tx_uart(c);
 9   //-------------------------[ VARIANT-3 ]---------
10   while (!empty_rx_uart()) {
11     char c = read_rx_uart();
12     assert (c != '#'); // assume sender filters '#'
13     write_tx_uart(c);
14   }
15   //-------------------------[ END ]---------------
16 }
```

Listing 3.4.: Example embedded software to illustrate overlay design and refinement.

to a TX FIFO. The implementation is presented in three different variants, from simple to more sophisticated. The driver implementation is interrupt-driven and triggered whenever the RX FIFO receives new data. Listing 3.5 shows the corresponding symbolic overlay for the UART to support the respective variant of the software driver. For clarity and brevity, we employ a slight pseudocode notation (in particular regarding the `assume` function that adds symbolic constraints) and omit irrelevant details (such as the constructor). In order to make use of overlays, we propose a methodology where the overlay is iteratively refined. For this purpose, we start with a coarse overlay and refine it on demand as necessary to support the tested embedded software.

The first software variant simply copies a single character from the RX to the TX FIFO (Line 3 - Line 4 in Listing 3.4). To support such a scenario, the overlay simply returns an arbitrary value on a read access (Line 10 in Listing 3.5). For the next variant, we assume a scenario where the received characters are constrained according to an environment model, i.e. we assume that the character '#' is never received. With variant 1 of the overlay, the variant 2 software would hit a spurious error because the assertion in Line 7 should not fail under the current environment assumptions. In order to support such a scenario, the outgoing UART data can be constrained accordingly by the overlay

```cpp
class UARTOverlay : public sc_core::sc_module {
  UART &uart; // reference to concrete UART
  // ...omit constructor...
  void transport(tlm::tlm_generic_payload &trans,
                 sc_core::sc_time &delay) {
    // process symbolic extension
    auto addr = trans.get_address();
    if (addr == RX_ADDR) {
//-------------------------[ VARIANT-1 ]----------
      auto data = SymbolicUint32();   // any value
//-------------------------[ VARIANT-2 ]----------
      auto data = SymbolicUint32();   // any value
      assume ((data & 0xff) != '#');  // avoid '#'
//-------------------------[ VARIANT-3 ]----------
      auto data = SymbolicUint32();   // any value
      assume ((data & 0xff) != '#');  // avoid '#'
      if (uart.empty()) {
        assume (data & (1 << 31));    // empty
      } else {
        assume (!(data & (1 << 31))); // not empty
      }
//-------------------------[ END ]----------------
      // pack the symbolic value into a TLM extension
      auto ext = new SymbolicExtension(data);
      trans.set_extension(ext);
    }
    // call the normal UART
    isock->b_transport(trans, delay);
  }
};
```

Listing 3.5.: Example overlay to illustrate overlay design and refinement.

(Line 13), as shown in variant 2 of the overlay implementation. The software variant 3 adds another layer of complexity (Line 10 - Line 14). Now, the copy process continues until the RX FIFO is empty (which makes this driver more generic).[19] Using variant 2 of the overlay, the software could spin infinitely in the copy loop because the empty bit is unconstrained symbolic and therefore independent of the actual UART status. This problem is fixed by the variant 3 refinement (Line 17 - Line 21) where an additional check is added to ensure that the underlying UART is not empty. As such, this last variant also demonstrates the benefits of having access to the actual peripheral from the overlay. Furthermore, the last variant covers the main functionality from the receive part of the UART which is the essential functionality regarding a symbolic analysis, as received data is potentially untrusted and needs to be properly processed by the software without it inhibiting unwanted behavior. The remaining omitted functionality is mainly responsible for configuring the UART and processing interrupts.

The motivational example demonstrates that symbolic overlays can be compact and yet enable comprehensive symbolic exploration of embedded firmware that interacts extensively with a peripheral (such as a UART) on a low abstraction level. From a design flow perspective, the example also indicates that peripheral models and overlays can be developed side by side, which eases maintenance and testing. That is, in a VP-based design flow (see Section 2.1), the concrete peripheral models can still be used normally in the concrete VP and changes to peripheral models are immediately available in both the concrete and the symbolic VP.

3.2.3. Evaluation

In order to evaluate our overlay mechanism and the proposed methodology for overlay refinements, we can conduct a case study with an example application for the RIOT [9] operating system. This application interacts with a sample sensor peripheral—modeled in SystemC TLM—for which an overlay is created and iteratively refined as part of the case study. RIOT is a popular multithreaded operating system for programming constrained devices; in this domain, prior work even considers it to be the “most prominent open source operating system” [79, p. 732]. Furthermore, the RIOT supports the SiFive HiFive1 hardware platform modeled by SymEx-VP. Therefore, we can conduct a symbolic analysis of RIOT-based firmware using the overlay-enhanced version of SymEx-VP presented in Subsection 3.2.2.

[19]We assume that the TX FIFO has a sufficiently large buffer and processes characters fast enough.

In the following, we present more details on the exemplary RIOT application (Subsection 3.2.3.1), then discuss our test setup (Subsection 3.2.3.2), and lastly present the obtained results (Subsection 3.2.3.3).

3.2.3.1. RIOT-based Example Application

For our case study, we consider an exemplary interrupt-driven application that periodically processes sensor data. The sensor data is generated by a custom TLM sensor model; the modeled sensor triggers an interrupt whenever new sensor data is available. The returned sensor data conforms to a filter, which is a memory-mapped register of the sensor that is configured by the firmware. A producer-consumer scheme is employed by the application to process this data. For this purpose, two RIOT threads are used which communicate with each other through RIOT's interprocess communication mechanism. The producer thread reads data from the sensor and passes received data to the consumer thread which processes the data and ultimately writes it to the UART. Assertions are used to perform sanity checks on the data, for instance, to check that sensor values are within a certain expected range. Considering that RIOT is a multithreading operating system, we believe that these kinds of producer-consumer patterns are common in embedded firmware that utilizes RIOT.

3.2.3.2. Test Setup

In order to ensure termination of the testing process, as the application is interrupt-driven and hence non-terminating, we bounded the number of processing iterations in the application. The testing process continues until all bugs are fixed. Since we start with coarse overlays that over-approximate the behavior of the real peripherals, spurious bugs are expected. Whenever encountering a spurious bug, a refinement of the overlays is necessary.

For our case study, an overlay is only required for the custom sensor peripheral (it acts as the only input device). Overlays for other SiFive HiFive1 peripherals are not necessary; they remain concrete. This includes the UART, since it only acts as an output device, but also other peripherals which are accessed by RIOT during the operating system initialization phase. Therefore, we only iteratively refine the custom sensor peripheral overlay according to the methodology introduced in Subsection 3.2.2.2. Contrary to the motivational example in Subsection 3.2.2.2, the tested firmware itself is not refined; it is assumed to be production-ready.

Table 3.3.: Evaluation results for different overlay refinements.

Iteration	#Instrs	Runtime	#Paths	Bug
I1	127 477	1.11 s	1	SB
I2	1 370 415	13.58 s	13	RB
I3	341 805 972	170 min	3270	-

3.2.3.3. Test Results

Table 3.3 presents the results of our conducted case study. The first column shows the overlay refinement iteration. The next three columns show the number of executed instructions (column: *#Instrs*), overall runtime (column: *Runtime*), and number of discovered execution paths (column: *#Paths*). Please note that, by using symbolic expressions, symbolic execution can cover a large set of different inputs on a single path. The last column indicates if a bug was found and, if so, if the bug was a spurious (SB) or a real bug (RB). In total, three refinement iterations (I1, I2, and I3) were performed until no more bugs were detected (in I3). All experiments were performed on a Linux system with an Intel i7-8565U processor.

In I1 the sensor overlay returns fully unconstrained data. While this abstraction is too coarse (since the sensor model filters data according to a configuration setting), we left it for demonstration purposes. As expected, a spurious bug (in the form of an assertion failure in the tested firmware) is detected fast; in fact, on the first path after executing around 127K instructions. We refined the sensor overlay accordingly to consider the filter setting of the modeled sensor, resulting in I2. Now a real application bug is detected (caused by a false assumption regarding potential values returned by the sensor in the application code) after 13 paths and around 1.4M instructions. For I3, we fixed the aforementioned bug and discovered no further bugs. In total 3270 paths were explored in I3 with around 341M executed instructions in 170 minutes. The conducted case study demonstrates that our proposed approach can be effective for finding bugs in embedded firmware that interacts on a low abstraction level with the hardware.

3.2.4. Related Work

Handling of peripheral interactions during dynamic testing (e.g. symbolic execution) of embedded firmware is an active and ongoing subject of research [207, Section 8.1.1]. The overlay approach, presented in this section, is conceptually similar to prior work on rehosting where peripheral behavior is approximated and these approximations are iteratively refined [77, 35, 68]. A survey of prior work on rehosting is provided by Fasano et al. [66]. However, rehosting approaches commonly operate on a higher abstraction level and do not use a modeling standard such as SystemC. Initial work towards facilitating SystemC peripheral models was done by Herdt et al., but this prior work requires manual adaptation of such models and does not support the SystemC simulation kernel [84]. Similarly, there is prior work that uses models manually extracted from the QEMU [16] simulator [90]. Related work has also introduced instruction-level models of software-visible hardware behavior for formal security verification [91]. Lastly, prior work in the hardware/software co-verification domain has experimented with operating on an even lower abstraction level by making use of Verilog [58] hardware models [129]. In this context, symbolic execution is also increasingly used to test hardware instead of software designs, both at the RTL [143, 1] and VP level [119, 86]. However, to the best of our knowledge, an approach that directly integrates symbolic execution with SystemC-based VPs, for the purpose of testing embedded software, is not available.

3.2.5. Conclusion

In conclusion, we have presented a TLM overlay mechanism to ease the integration of existing SystemC TLM hardware models with symbolic execution. To this end, the conducted case study demonstrates the applicability and effectiveness of our proposed approach for this purpose. In future work, we want to further ease the adoption of existing models by investigating automated methods to derive peripheral overlays from existing SystemC TLM peripherals. A starting point in this regard might be a half-automated approach that leverages peripheral interface descriptions. Additionally, we plan to devise automated methods to refine overlays or localize the exact source of a spurious error. Presently, this is a manual procedure, although employment of the existing VP-based debug infrastructure turned out to be advantageous for this purpose. Compared to the modeling approach presented in Section 3.1, we envision that TLM overlays will further ease adoption of our symbolic execution approach.

Chapter 4.

Formal ISA Semantics for Extending the Analysis

Chapter 3 focused on environment modeling for embedded firmware using the SystemC hardware modeling language. For this purpose, the chapter contributed an integration of VPs with symbolic execution. While the chapter was centered around modeling of hardware peripherals using SystemC, the combination with VPs also required an integration of symbolic execution with the ISS of the utilized VP (see Subsection 3.1.2.3). The ISS is responsible for executing binary code instructions specified by the ISA of the targeted hardware platform. As explained in Section 2.3, symbolic execution requires custom software interpretation and, therefore, a custom implementation of the ISA (i.e. a custom variant of the ISS). In Subsection 3.1.2.3, a symbolic ISS for the RISC-V architecture was presented. This ISS was written in C++ and obtained through manual modifications of the existing concrete and manually-written ISS provided by `riscv-vp` [85].

Unfortunately, manual creation of a symbolic ISS makes it challenging to extend the proposed symbolic execution approach to additional instructions. However, especially in the embedded domain, it is paramount to be able to easily extend the analysis to additional instructions as—due to the tight integration of hardware and software in this domain—embedded systems often use domain-specific instructions for optimization purposes [51, Section 2]. Presently, the symbolic semantics need to be specified manually in C++ for each new instruction which is a laborious process and an obstacle for extensibility. This also becomes apparent when one considers the possibility of employing the proposed symbolic execution approach for other mainstream ISAs such as ARM. Compared to RISC-V, these architectures have an enormous complexity. For example, the ARMv8-A ISA specification consists of 6300 pages of mostly natural language [4]. Implementing a symbolic ISS manually requires reading and understanding the instruction semantics specified in this document and then manually translating this specification to C++ code. Naturally, this is a cumbersome process that is error-prone and requires additional verification and testing effort to ensure correctness of the implementation. Correctness of the symbolic ISS is paramount as otherwise the symbolic execution may miss paths through the tested firmware, which in turn means that vulnerabilities may remain undetected and could thus be exploited once the firmware is deployed in a production environment. Lastly, for the design of new embedded systems, the symbolic execution approach proposed in Chapter 3 also requires maintaining two variants of the VP's ISS: a concrete one for software development and a symbolic one for software testing using symbolic execution. This creates additional development overhead and may negatively impact the time-to-market of a new embedded system (see Section 2.1).

The underlying problem of the outlined challenges is the manual translation of the ISA specification to an executable ISS that implements this specification. This process is manual because the specification is given in natural language, which is not machine-readable as it can be ambiguous. In order to overcome this challenge, a large body of prior work has proposed giving ISA specifications in formal instead of natural language, i.e. language that is unambiguous and therefore machine-readable [57, 150, 5, 161, 26]. The formal language itself describes the ISA semantics in terms of several language primitives. The most comprehensive approach in this regard is Sail which formally describes a wide range of different architectures (ARM, RISC-V, MIPS, IBM Power, and x86) using the same language primitives [5]. Sail has even been adopted as the official formal model for the RISC-V architecture. In general, RISC-V has sparked a lot of research on formal specification due to its open and modular nature. Different formal specifications for different purposes exist for this architecture; a survey of existing formal RISC-V specifications was conducted by the RISC-V Foundation in 2019 [152].

Naturally, formal ISA specifications can be used for all kinds of purposes, including code generation, formal verification, documentation, and simulation. In this chapter, we investigate the use of formal ISA specifications for symbolic execution of binary code, thereby addressing the outlined shortcomings of the symbolic execution approach presented in Chapter 3. We contribute a novel executable RISC-V formal model that is specifically tailored to the creation of binary analysis tools in the lineage of prior work on modular interpreters [116, 177, 93]. Based on this new formal model, we propose a symbolic execution approach that achieves extensibility by building a symbolic ISS on top of the language primitives provided by this formal model. Lastly, we propose an integration of this enhanced symbolic execution approach with our prior work on modeling of hardware peripherals using VPs (Chapter 3) through C/C++ code generation.

4.1. A Flexibel Formal Model for the RISC-V ISA

This section describes a flexible framework for the creation of binary analysis tools that can be extended to additional instructions and retargeted to other ISAs. This is achieved by building these analysis tools on top of a formal description of ISA semantics. Prior work has already proposed the use of formal descriptions for this purpose [117]. However, building a symbolic execution engine on top of formal descriptions is challenging as symbolic execution requires a flexible value representation within the formal language. That is, it cannot be assumed that instructions operate on fixed-width concrete integers and therefore it is insufficient to only provide a formal description of the concrete operational semantics. Otherwise, it is not possible for instruction operands to be represented as symbolic values.

In order to overcome this challenge, we contribute a highly flexible and versatile model of the RISC-V ISA in the functional programming language Haskell[1]. As opposed to existing models [117, 161, 22], the interpretation of the ISA and the representation of instruction operands can be varied. To this end, we define a Haskell-based Embedded Domain-Specific Language (EDSL) via a free monad construction that facilitates a custom expression language to express operations on instruction operands (see Subsection 4.1.1.2). The idea is that the free monad models the computation given by a sequence of operations from the ISA, where the model of computation (i.e. the interpretation) can be varied, from simple state transitions that simulate the ISA faithfully to custom analyses. Furthermore, the custom expression language enables us to vary the representation of instruction operands, thereby also enabling the implementation of symbolic execution as an instantiation of a custom analysis. To the best of our knowledge, our approach is the first that enables the creation of symbolic execution engines as custom interpreters for a formal ISA model. We motivate the advantages of this approach by modeling a simple ISA first (Subsection 4.1.2.1) and then extend this approach to the real-world RISC-V architecture (Subsection 4.1.3). Afterward, we conduct a case study with dynamic information flow tracking (Subsection 4.1.3.3) to illustrate the implementation of custom ISA interpreters and evaluate the performance (i.e. simulation speed) of our implementation (Subsection 4.1.4). Based on the presented formal model, we contribute a novel symbolic execution engine in Section 4.2.

[1]`https://haskell.org`

4.1.1. Preliminaries

In the following, we provide background information on formal ISA models and the free monad abstraction which we use for our proposed formal model of the RISC-V architecture.

4.1.1.1. Formal ISA Models

Traditionally, ISAs have been specified in natural language. However, as natural language is ambiguous and thus not machine-readable, it has become popular to specify the ISA in a formal language—one which has unambiguous semantics—thereby obtaining a machine-readable formal ISA model. Formal models can be used for all kinds of purposes, including verification, testing, documentation, and simulation [152]. Considering these different use cases, formal models come in a variety of languages (and we give a more comprehensive overview in Subsection 4.1.5), but we can broadly distinguish between the following:

1. Models using a Domain-Specific Language (DSL), designed specifically for this purpose, such as Sail [5].

2. Models using a universal non-executable modeling language, such as the logic supported by a theorem prover.

3. Models using a universal, general-purpose (often functional) programming language.

For our work, we chose the third approach, as DSLs (such as Sail) come with their own custom-built tooling, which does not support our use case well. Furthermore, their formal description is often abstract, which means they are not as concise or convenient to work with as models written in general-purpose programming languages. Similarly, contrary to general-purpose programming languages, theorem provers do not provide much programming support. Instead, they allow reasoning about models (proving properties such as correctness, completeness, and so on). For this reason, we chose a formal model in a functional programming language (namely Haskell) as the basis of our work. This allows us to model the ISA at a high level of abstraction, while at the same time providing access to a rich ecosystem of existing libraries to process the ISA model conveniently and in a flexible, modular fashion. For example, for the symbolic execution engine presented in Section 4.2, we make use of an existing Haskell wrapper for the Z3 [125] SMT solver library. However, should we in the future choose to expand on proofs and correctness, existing work to either generate Haskell from a proof assistant (e.g. Coq [18]) or import Haskell into a proof assistant would make this possible, so this option remains open for future work.

4.1.1.2. Free Monads and EDSLs

Conceptually, an ISA is essentially a low-level imperative programming language with predefined bit-vector data types (words of a given length). The semantics of imperative programs covers different aspects (stateful computations, continuations, exceptions, etc.) each of which can be modeled in Haskell using monads [177, 116, 115]. Combining these monads is a notoriously tricky exercise. Early work on interpreting imperative programs used monad transformers for this effect [116], but more recent work uses free monads for better performance and extensibility (see Subsection 4.1.5.2 for a detailed comparison).

Free monads enable the creation of EDSLs [92], i.e. DSLs which are embedded into an existing (often functional) general-purpose programming language.[2] Since EDSLs are tightly integrated with a chosen host language—Haskell in our case—they allow reuse of existing functionality provided by this host language, thereby easing the implementation of DSLs. Similar to the aforementioned prior work on the semantics of imperative programs, we facilitate free monads to create an EDSL for describing ISA semantics.

When implementing an EDSL through free monads, the basic operations of the EDSL are defined through type constructors. The free monad allows for the composition of these type constructors, thus yielding an embedded language that supports the defined operations. The category-theoretic construction of free monads was given by Kelly [102], and first described in the context of functional programming by Swierstra [179]. In our implementation, we use an enhanced version of this concept as implemented by the `freer-simple`[3] Haskell library. More information on this enhanced version of the concept is provided in publications by Kiselyov et al. [105, 106].

4.1.2. Modeling an ISA

We motivate our approach and its advantages by applying it to an intentionally simple ISA. The ISA implements a 32-bit load-store architecture with five instructions; each of these can be thought of as representing a class of similar instructions in a real ISA:

[2]Functional languages are popular for this purpose because of their declarative nature.
[3]`https://hackage.haskell.org/package/freer-simple`

1. `LOADI` *imm reg*: Load immediate into register *reg*.
2. `ADD` *dst src1 src2*: Add two registers into *dst*.
3. `LW` *dest addr*: Load word from memory at *addr* into register *dest*.
4. `SW` *addr src*: Store word from register *src* into memory at *addr*.
5. `BEQ` *reg1 reg2 off*: Relative branch by *off* if registers *reg1* and *reg2* are equal.

The ISA supports 16 general-purpose registers, word-addressable memory, and a program counter which points to the current instruction in memory. All registers and memory values are 32-bit wide and treated as signed values by all instructions. Instruction fetching and decoding are not discussed. The instruction set is modeled as a Haskell data type (where `Word` and `Addr` are type synonyms for 32-bit integers):

```haskell
newtype Reg = Reg { reg :: Int } deriving (Ord, Eq)
data INSTR
    = LOADI Word Reg
    | ADD Reg Reg Reg
    | LW Reg Addr
    | SW Addr Reg
    | BEQ Reg Reg Word
```

4.1.2.1. A First Model

The execution model formally describes how instructions are executed. It specifies the system state, and how instructions change the system state (including the control flow). Listing 4.1 provides a simple Haskell execution model for our exemplary ISA. The architectural state `System`, upon which instructions are executed, is a tuple consisting of two finite maps for the memory and register file as well as a concrete program counter. Instruction execution itself is implemented as a pure function which performs a pattern match on the instruction type and returns a new system state, embedded into a state monad (`State System a`).

Unfortunately, this primitive ISA model has multiple shortcomings. Consider a simple software analysis task for which we want to extend our model to track the number of memory accesses during program execution. For this, we merely need to extend the system state with an access counter, and increment the counter whenever memory access takes place (operations `LW` and `SW`). A possible implementation of this modification is

```haskell
type System = (Registers
               , Mem
               , ProgramCounter)


execute :: INSTR -> State System ()
execute i = modify $
  \(regs, mem, pc) -> case i of
    LOADI imm r -> (insert r imm regs,
                    mem, nextInstr pc)
    ADD rd rs1 rs2 -> let
         v1 = regs ! rs1
         v2 = regs ! rs2
        in  (insert rd (v1+v2) regs,
             mem, nextInstr pc)
    LW r addr -> let
         w = mem ! addr
        in  (insert r w regs, mem,
             nextInstr pc)
    SW addr r -> let
         v = regs ! r
        in (regs, insert addr v mem,
            nextInstr pc)
    BEQ r1 r2 off -> let
         v1 = regs ! r1
         v2 = regs ! r2
         br = if v1 == v2
           then pc+off
           else nextInstr pc
        in (regs, mem, br)
```

Listing 4.1.: Concrete Haskell model of our simple ISA from Subsection 4.1.2.

```haskell
type System' = (Registers
                , Mem
                , ProgramCounter
                , Int)

execute' :: INSTR -> State System' ()
execute' i = modify $
  \(regs, mem, pc, counter) -> case i of
    LOADI imm r -> (insert r imm regs, mem,
                    nextInstr pc, counter)
    ADD rd rs1 rs2 -> let
         v1 = regs ! rs1
         v2 = regs ! rs2
        in (insert rd (v1+v2) regs,
            mem, nextInstr pc, counter)
    LW r addr -> let
         w = mem ! addr
        in (insert r w regs, mem,
            nextInstr pc, succ counter)
    SW addr r -> let
         v = regs ! r
        in (regs, insert addr v mem,
            nextInstr pc, succ counter)
    BEQ r1 r2 off -> let
         v1 = regs ! r1
         v2 = regs ! r2
         br = if v1 == v2
           then pc+off
           else nextInstr pc
        in (regs, mem, br, counter)
```

Listing 4.2.: Memory accesses analysis through modifications of Listing 4.1.

shown in Listing 4.2. Note how, even though our extension to the previous solution did not modify the control flow of the program in any way, we still had to restate the control flow for all supported instructions of our ISAs. For our small ISA, this inconvenience seems feasible, but considering that a real ISA often has more than 80 instructions, the task of modifying the execution becomes cumbersome and error-prone.

Hence, our aim is to give a modular, abstract representation of ISA semantics, based upon which we can then implement software analysis techniques that require a different kind of interpretation with minimal effort. Such techniques may include symbolic execution [10] or dynamic information flow tracking [178].

4.1.2.2. Our Approach

In summary, the problem with the previously outlined approach is that the model of the semantics (a state transition given by a state monad) is given in a concrete and monolithic form: there is no separation between the different aspects of the semantics. However, the semantics of an ISA has multiple aspects: memory access, register access, arithmetic, and control flow, and most analyses only concern one or two of them (e.g. memory access, or arithmetic). Yet, if we want to change the representation of the state, this affects all operations; similarly, if we wish to reason about, e.g. integer arithmetic to show the absence of integer overflow, we need to re-implement all operations.

Thus, we intend to give the semantics of our ISA by combining constituting parts, which we can change individually. To this end, we define an EDSL which represents the operations of an abstract machine implementing the ISA, e.g. loading and storing words into registers, using a free monad as introduced in Subsection 4.1.1.2.

The operations comprising the EDSL are given by a parameterized type `Operations`, see Listing 4.3.[4] The `Operations` data type models the ISA in abstract terms; the free monad `Free Operations` describes combinations of these, which are an abstract representation of the control flow of a (sequence of) ISA operations. This representation is given by a function `controlFlow :: INSTR -> Free Operations ()`, which defines the control flow for a given instruction (see Listing 4.4); by composing these, we obtain the control flow for a program (the sequence of operations). To reconstruct the concrete execution of the ISA instructions from the previous section (Listing 4.1), we need to map the operations in the free monad to concrete monadic effects, in our case in Haskell's pure `State` monad. An example implementation of a function which performs this mapping is

[4]For convenience, we add a smart-constructor for each constructor of the data type.

```haskell
data Operations r
    = LoadRegister Reg (Word -> r)
    | StoreRegister Reg Word r
    | IncrementPC Word r
    | LoadMem Addr (Word -> r)
    | StoreMem Addr Word r
    deriving Functor

loadRegister :: Reg -> Free Operations Word
loadRegister r = Free (LoadRegister r Pure)

storeRegister :: Reg -> Word -> Free Operations ()
storeRegister r w = Free (StoreRegister r w (Pure ()))

incrementPC :: Word -> Free Operations ()
incrementPC v = Free (IncrementPC v (Pure ()))

loadMem :: Addr -> Free Operations Word
loadMem addr = Free (LoadMem addr Pure)

storeMem :: Addr -> Word -> Free Operations ()
storeMem addr w = Free (StoreMem addr w (Pure ()))
```

Listing 4.3.: Overview of provided EDSL operations for describing the ISA.

```haskell
controlFlow :: INSTR -> Free Operations ()
controlFlow = \case
    LOADI imm r -> storeRegister r imm >> incrementPC instrSize
    ADD rd r1 r2 -> do
        v1 <- loadRegister r1
        v2 <- loadRegister r2
        storeRegister rd (v1+v2)
        incrementPC instrSize
    LW r addr -> do
        v <- loadMem addr
        storeRegister r v
        incrementPC instrSize
    SW addr r -> do
        v <- loadRegister r
        storeMem addr v
        incrementPC instrSize
    BEQ r1 r2 off -> do
        v1 <- loadRegister r1
        v2 <- loadRegister r2
        if v1 == v2 then incrementPC off else incrementPC instrSize
```

Listing 4.4.: Interpreting an in ISA instruction in the free monad.

```haskell
execute :: State -> Free Operations () -> State
execute st = flip execState st . iterM go where
    go = \case
        LoadRegister reg f -> gets (\(rs,_,_) -> rs ! reg) >>= f
        StoreRegister reg w c ->
            modify (\(rs, mem, pc) -> (insert reg w rs, mem, pc)) >> c
        IncrementPC w c -> modify (\(rs,mem,pc) -> (rs,mem,pc+w)) >> c
        LoadMem addr f  -> gets (\(_,mem,_) -> mem ! addr) >>= f
        StoreMem addr w c ->
            modify (\(rs,mem,pc) -> (rs, insert addr w mem, pc)) >> c
```

Listing 4.5.: Evaluating the control flow using the State monad.

provided in Listing 4.5. Since we have now separated the control flow and semantics of effects, we could also use any other (monadic) effects for the evaluation without changing the control flow. Reconstructing the example from Listing 4.2 just requires adjustments in the semantics without restating the control flow as shown in Listing 4.6 (performed adjustments are highlighted using underlined text).

While this is a major advantage in terms of reusability, there is still room for improvement. In particular, we are not able to change the semantics of the expression-level calculations an operation performs, since the data type of our EDSL assumes concrete types, which entails they are already evaluated. Hence, we generalize our `Operations` type to allow a representation of the evaluation of expressions, much like we did for the instructions (except that the evaluation of expressions is not monadic and hence we do not need a free monad here). For that, we need to introduce a simple expression language that replaces all the constant values, e.g. the constructor `StoreRegister :: Reg -> Word -> r` becomes `StoreRegister' :: Reg -> Expr w -> r`. Additionally, we need to adjust the `Operations` type so that it becomes polymorphic in the word type.

Listing 4.7 shows the necessary changes, e.g. the `execute'''` function is now provided with an expression-interpreter `evalE`, which is used to evaluate expressions generated by the control flow. The `Operations` are now polymorphic in the word type and the semantics of the internal computations can be changed by adjusting `evalE`; this allows our approach to be used to implement software analysis techniques on the ISA level. In the next section, we will present an application of our approach to the RISC-V ISA, and utilize the resulting RISC-V model to implement one exemplary software analysis technique as a case study.

```haskell
execute' :: State'' -> Free Operations () -> State''
execute' st = flip execState st . iterM go where
    go = \case
        LoadRegister reg f -> gets (\(rs,_,_,_) -> rs ! reg) >>= f
        StoreRegister reg w c -> modify
            (\(rs, mem, pc, counter) ->
                (insert reg w rs, mem, pc, counter)) >> c
        IncrementPC w c -> modify
            (\(rs, mem, pc, counter) -> (rs, mem, pc+w, counter)) >> c
        LoadMem addr f  -> do
            v <- gets (\(_,mem,_, counter) -> mem ! addr)
            modify (\(rs, mem, pc, counter) -> (rs, mem, pc, succ counter))
            f v
        StoreMem addr w c ->
            modify (\(rs, mem, pc, counter) ->
                (rs, insert addr w mem, pc, succ counter)) >> c
```

Listing 4.6.: Executing and counting memory accesses.

4.1.3. Modeling the RISC-V ISA

As an application of our approach, we created an abstract model of the RISC-V ISA. RISC-V is an emerging Reduced Instruction Set Computer (RISC) architecture which has recently gained traction in both academia and industry. As explained in Section 2.2, RISC-V is developed as an open standard free from patents and royalties. It is designed in a modular way: the architecture consists of a base instructions set and optional extensions (e.g. for atomic instructions) which can be combined as needed [153].

We refer to our model of the RISC-V architecture as LibRISCV. As the name suggests, LibRISCV is a Haskell library that can be used to implement different interpreters for RISC-V software. As such, the library provides an instantiable framework for versatile interpretation of RISC-V software in binary form. Figure 4.1 illustrates how the concepts from Subsection 4.1.2.2 are applied to RISC-V to achieve versatile interpretation. The figure will be further described in the following subsections.

4.1.3.1. Instruction Decoder

As depicted in Figure 4.1, our RISC-V implementation receives binary code as an input value. This binary code constitutes RISC-V machine code and is converted to an alge-

```haskell
data Expr a = Val a | Add (Expr a) (Expr a) | Eq (Expr a) (Expr a)

data Operations' w r
    = LoadRegister' Reg (Expr w -> r)
    | StoreRegister' Reg (Expr w) r
    | IncrementPC' (Expr w) r
    | LoadMem' Addr (Expr w -> r)
    | StoreMem' Addr (Expr w) r

evalE :: Expr Word -> Word
evalE = \case
    Val a -> a
    Add e e' -> evalE e + evalE e'
    Eq e e' -> if evalE e' == evalE e then 1 else 0

execute''' :: (Expr Word -> Word) -> State''
              -> Free (Operations' Word) () -> State''
execute''' evalE st = flip execState st . iterM go where
    go = \case
        LoadRegister' reg f -> gets (\(rs,_,_,_) -> Val $ rs ! reg) >>= f
        StoreRegister' reg w c -> modify
            (\(rs, mem, pc, counter) ->
                (insert reg (evalE w) rs, mem, pc, counter)) >> c
        IncrementPC' w c -> modify
            (\(rs,mem,pc,counter) -> (rs,mem,pc+ evalE w, counter)) >> c
        LoadMem' addr f -> do
            v <- gets (\(_,mem,_, counter) -> mem ! addr)
            modify (\(rs,mem,pc,counter) -> (rs, mem, pc, succ counter))
            f $ Val v
        StoreMem' addr w c -> modify (\(rs,mem,pc,counter)
            -> (rs, insert addr (evalE w) mem, pc, succ counter)) >> c
```

Listing 4.7.: Operations type with simple expression language.

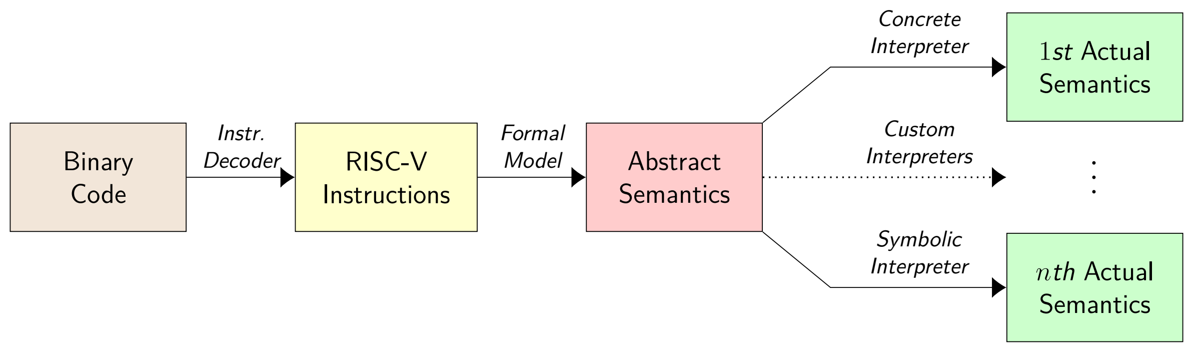


Figure 4.1.: Application of our ISA modeling approach to the RISC-V architecture.

braic data structure representing instructions mandated by the RISC-V standard using an instruction decoder. Contrary to imperative programming languages, execution and decoding/parsing are heavily intertwined for machine code. We can only decode the next instruction after finishing execution of the current instruction (fetch-decode-execute cycle) [175, Section 14.3]. For example, when executing a branch instruction, the next fetched instruction depends on the result of the branch. We make use of lazy evaluation to model the fetch-decode-execute cycle as part of our control flow description. That is, the fetching of the next instruction is itself—non-strictly—modeled, using free monads as outlined in Subsection 4.1.2.2.

Contrary to existing work on formal models (e.g. Sail [5]), a description of RISC-V instruction decoding is not part of our EDSL. Instead, the LibRISCV instruction decoder is automatically generated from `riscv-opcodes`[5], an existing formal language which describes how binary code is mapped to RISC-V instructions (without modeling instruction semantics). Based on algebraic data types, returned by the instruction decoder, we specify the *abstract semantics* of RISC-V instructions through a formal ISA model described in the following.

4.1.3.2. Formal Model

An overview of the ISA model provided by LibRISCV is available in Figure 4.2. As illustrated in Figure 4.1, the central component of the formal model is the description of the abstract instruction semantics, which represents the lazily-generated control flow of the RISC-V ISA operations. As discussed in Subsection 4.1.2.2, we use free monads

[5]`https://github.com/riscv/riscv-opcodes`

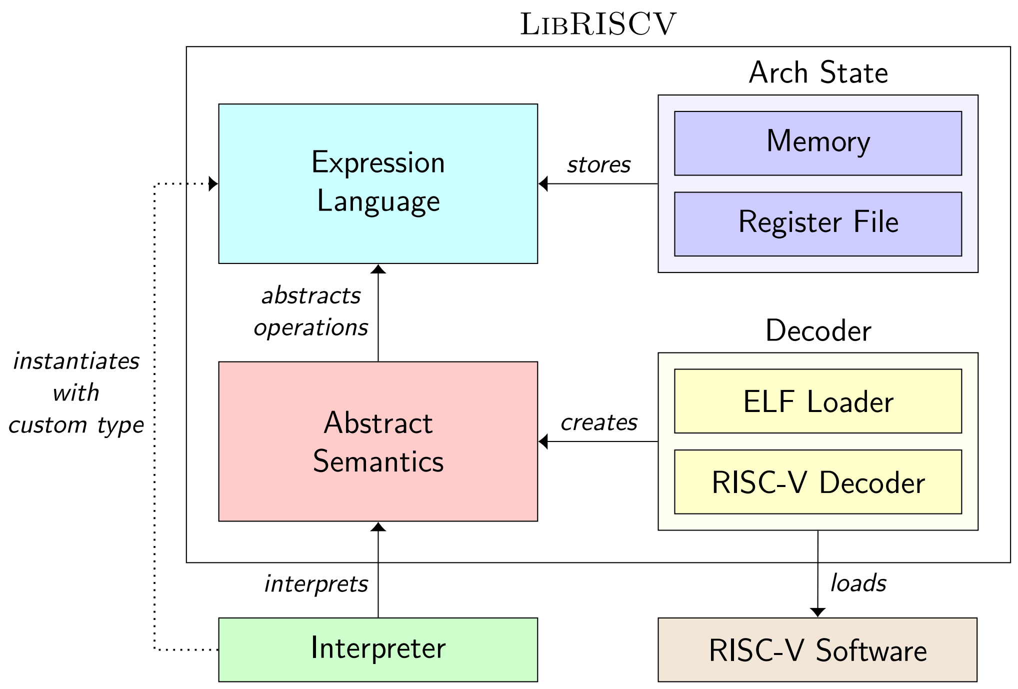


Figure 4.2.: Overview of the RISC-V ISA model provided by LibRISCV.

for this purpose. For the implementation of free monads, we use the `freer-simple` library. The library provides an improved implementation of the free monad approach referenced in Subsection 4.1.1.2. Within the abstract description of instruction semantics, all operations on register/memory values are abstracted using a generic expression language. The expression language is implemented as an algebraic data type with an associated evaluation function, as illustrated in Listing 4.7. The algebraic data type, used by the expression language, is parameterized over a custom type. The architectural state (i.e. memory and register file) is also parameterized over this type. As shown in Figure 4.2, the abstract description of instruction semantics is based on an instruction type that is generated by the aforementioned instruction decoder. The decoder is responsible for loading RISC-V software in ELF binary form and for decoding/parsing instruction words—contained in the ELF file—according to the RISC-V specification.

Based on the abstract semantics, we can provide different interpreters which implement the *actual semantics* for decoded RISC-V instructions as illustrated in Figure 4.1, such as concrete or symbolic execution of modeled instructions. The actual semantics implement the state transition for each modeled instruction while the abstract semantics only describe the control flow. Each interpreter instantiates the expression language with a type. Based on this type, an interpreter for the formal ISA model (i.e. the expression language and the free `Operations` monad) needs to be supplied. Presently, LibRISCV provides a formal model for the 32-bit variant of the RISC-V base instruction set (40 instructions). Based on this formal model, we have implemented a concrete interpreter for RISC-V instructions. Both the model and the concrete interpreter are written in roughly 1500 LOC and can be obtained from GitHub.[6] Using the concrete interpreter, we were able to successfully execute and pass the official RISC-V ISA tests for the 32-bit base instruction set.[7] These tests include multiple test programs (one for each instruction) which check if the implemented behavior of an instruction conforms to the specification. Passing these tests indicates that our model correctly captures the semantics of the base instruction set. In the following, we illustrate how custom interpreters—beyond the standard concrete interpretation—can be implemented on top of our abstract model, thereby making use of its flexibility.

4.1.3.3. Custom Interpreters

Our model of the RISC-V ISA is designed for maximum flexibility and versatility, along the lines sketched in Subsection 4.1.2.2. This allows implementing different interpretations of the ISA on top of our abstract model with minimal effort. Conceptually, each custom interpreter implements actual semantics for the abstract semantics provided by the formal ISA model (see Figure 4.1). In order to implement a custom RISC-V interpreter, an evaluator for the expression language and an interpreter for the free `Operations` monad need to be provided. As an example, dynamic information flow tracking [178], where data-flow from input to output is analysed, can be implemented using the following polymorphic data type:

```haskell
data Tainted a = MkTainted Bool a

instance Conversion (Tainted a) a where
    convert (MkTainted _ v) = v
```

The product type `Tainted` tracks whether a value of type `a` is subject to data-flow analysis. Furthermore, a conversion to `Word32` is implemented through an instance declaration for the `Tainted` type. This conversion is the only class constraint imposed by our abstract model on the type used by the custom interpreter.[8]

[6] `https://github.com/agra-uni-bremen/libriscv`

[7] `https://github.com/riscv/riscv-tests`

[8] This constraint is necessary as the instruction decoder operates on `Word32` values.

A sample evaluator of the expression language for `Tainted Word32` looks as follows:[9]

```haskell
evalE :: Expr (Tainted Word32) -> Tainted Word32
evalE (FromImm t)   = t
evalE (FromInt i)   = MkTainted False $ fromIntegral i
evalE (AddU e1 e2) = MkTainted (t1 || t2) $ v1 + v2
    where (MkTainted t1 v1) = evalE e1; (MkTainted t2 v2) = evalE e2
```

The evaluator performs standard concrete integer arithmetic on the `Word32` encapsulated within the `Tainted` type. However, if one of the operands of the arithmetic operations is a tainted value, then the resulting value is also tainted. This enables a simple data-flow analysis for initially tainted values. Based on the evaluation function, an interpretation of the control flow is shown in the following, where $f \rightsquigarrow g$ denotes a natural transformation from f to g (as provided by the `freer-simple` library):

```haskell
type ArchState = ( REG.RegisterFile IOArray (Tainted Word32)
                 , MEM.Memory IOArray (Tainted Word8))

type IftEnv = (Expr (Tainted Word32) -> Tainted Word32, ArchState)

iftbehavior :: IftEnv -> Free Operations (Tainted Word32) ~~> IO
iftbehavior (evalE , (regFile, mem)) = \case
    (ReadRegister idx) -> REG.readRegister regFile idx
    (WriteRegister idx reg) -> REG.writeRegister regFile idx (evalE reg)
    (LoadWord addr) -> MEM.loadWord mem (convert $ evalE addr)
    (StoreWord addr w) -> MEM.storeWord mem (convert $ evalE addr)
                                            (evalE w)
```

This function operates on a polymorphic register and memory implementation. Expressions are evaluated using `evalE`, and then written to the register file or memory. When execution terminates, we can inspect each register and memory value to check whether it depends on an initially tainted input value. As shown, the interpreter only implements a subset of the `Operations` monad and the expression language; a complete implementation is provided in the `example/` subdirectory on GitHub.[10] Colleagues have already implemented dynamic information flow tracking for RISC-V machine code in prior work based on the `riscv-vp` simulator mentioned in Chapter 3 [142]. For this prior implementation, they had to modify `riscv-vp` extensively to allow for such an analysis to be

[9]The `FromImm`, `FromInt`, and `AddU` constructors belong to our expression abstraction.
[10]`https://github.com/agra-uni-bremen/libriscv/tree/tfp-2023/example`

performed. This was necessary because `riscv-vp` does not separate instruction semantics from instruction execution. In this context, the case study provided here serves to demonstrate that such techniques can be more easily implemented on top of an abstract formal model as custom interpreters for this model.

4.1.4. Performance Evaluation

Free monads introduce a well-known performance problem [105, Section 2.6]. As our approach is focused on implementing interpreters, simulation performance is important when executing real-world software. To evaluate simulation speed, we conduct a comparison with existing RISC-V simulators and specifically quantify the impact of the utilized `freer-simple` library on simulation performance. For this purpose, we leverage the existing Embench 1.0 benchmark suite [70]. Embench contains multiple benchmark applications which perform different computation-intensive tasks (e.g. checksum calculation). We compiled all applications for the 32-bit RISC-V base instruction set, executed them with different RISC-V simulators, and measured execution time in seconds. The results are shown in Table 4.1. All experiments have been conducted on an Intel Xeon Gold 6240 running an Alpine Linux 3.17 Docker image. Artifacts for the performed evaluation are available on Zenodo [183].

For each benchmark application in Table 4.1, we list the execution time in seconds for different RISC-V simulators. In order to specifically quantify the performance impact of the `freer-simple` library, we use a modified version of LibRISCV as a baseline where we manually removed the dependency on `freer-simple` and evaluate the ISA directly in Haskell's `IO`-monad. As such, this baseline version is conceptually similar to the primitive model presented in Subsection 4.1.2.1, i.e. the interpretation cannot be varied and it unconditionally performs concrete execution of RISC-V instructions. To contextualize the obtained results, we performed further experiments with existing Haskell implementations of the RISC-V ISA, namely Forvis [22] and GRIFT [161]. Contrary to our own work, these implementations do not utilize free monads (see Subsection 4.1.5). Lastly, Table 4.1 also contains evaluation results for the aforementioned `riscv-vp`, which is written in the C++ programming language [85]. To summarize benchmark results, Table 4.1 provides the geometric mean on a per-simulator basis in the bottom row.

Naturally, the C++ implementation (`riscv-vp`) has the lowest execution time over all benchmark applications. On average, it is roughly three times faster than our own Haskell implementation of the RISC-V ISA (LibRISCV). This is to be expected as, con-

Table 4.1.: Execution time comparison in seconds with existing RISC-V simulators.

Benchmark	baseline	libriscv	forvis	grift	riscv-vp
aha-mont64	21.68	41.32	53.81	351.85	14.15
crc32	8.71	16.61	21.08	148.69	5.75
cubic	28.80	57.99	71.90	614.11	19.20
edn	80.16	160.36	193.93	1 680.24	53.62
huffbench	8.31	15.41	20.18	108.62	5.60
matmult-int	41.71	82.72	96.94	820.24	28.07
minver	13.87	26.93	33.87	272.13	9.16
nbody	24.55	48.85	58.78	529.97	16.50
nettle-aes	8.91	16.19	19.99	118.77	5.93
nettle-sha256	6.94	12.50	15.68	89.82	4.43
nsichneu	4.19	7.63	9.30	59.18	2.79
picojpeg	13.99	26.30	38.66	203.55	9.73
qrduino	11.85	23.33	30.91	200.08	8.52
sglib-combined	7.94	14.33	18.52	106.38	5.24
slre	6.82	12.56	15.88	91.89	4.55
st	16.18	32.48	38.65	344.91	10.94
statemate	1.69	3.26	5.20	23.68	1.39
ud	14.44	27.10	33.47	222.35	9.42
wikisort	7.57	14.49	18.32	136.27	5.09
Geometric mean	12.07	23.02	29.37	197.96	8.16

trary to Haskell, C++ is not garbage collected. Nonetheless, and despite the employment of free monads, LibRISCV is—on average—still faster than Forvis and GRIFT. While LibRISCV and Forvis have similar execution time results, GRIFT is slower even though it is also written in Haskell. We attribute this to the fact that GRIFT represents the semantics as "symbolic expressions in a bitvector expression language" which is "suboptimal for fast simulation" [71]. The performance impact of the free monad abstraction (used in LibRISCV) can be estimated by comparing simulation performance with the baseline column in Table 4.1. As discussed above, the baseline column represents the execution time for a LibRISCV variant which does not use the `freer-simple` library. The gathered data indicates that LibRISCV is two times slower than the baseline version, confirming that free monads have a significant impact on simulation performance. Nonetheless, LibRISCV is still faster than existing Haskell implementations (Forvis and GRIFT) and approximately only three times slower than a primitive C++ implementation (`riscv-vp`). As such, we believe the induced performance penalty to be acceptable for our use case as the advantages of free monads outweigh this disadvantage by far.

4.1.5. Related Work

In the following, we discuss related work on formal ISA semantics, modular interpreters for imperative programming languages, and software analysis tools.

4.1.5.1. Formal Specifications

Formal semantics for ISAs is an active research area with a vast body of existing research. Specifically regarding RISC-V, a public review of existing formal specifications was conducted by the RISC-V foundation in 2019 [152]. From this review, Sail [5] emerged as the official formal specification for the RISC-V architecture. Sail is a custom DSL for describing different ISAs and comes with tooling for automatically generating simulators from this description. However, we believe a functional specification in a programming language like Haskell to be more suitable for rapid prototyping of custom interpreters. Similar to our own work, existing work on GRIFT [161], Forvis [22], and riscv-semantics [26] models the RISC-V ISA using a Haskell EDSL. Forvis and riscv-semantics are explicitly designed for readability and thus only use a subset of Haskell. As opposed to our own work, Forvis executes instructions directly and does not separate the description of instruction semantics from their execution. Prior work on riscv-semantics is closer to our own work and uses type classes to define the abstract semantics and then uses monads, implementing these type classes, to describe the actual semantics [26, Section 1]. Unfortunately, the central `RiscvMachine` type class of riscv-semantics is parameterized over a bit-width and can thus only be instantiated with concrete fixed-with integers [26, Figure 2]. Therefore, this prior work does not achieve the same flexibility as our own work on LibRISCV and hence cannot be used to implement symbolic semantics. Similarly, prior work on GRIFT uses a bit-vector expression language to provide a separate description of instruction semantics. However, GRIFT's expression language is also designed around natural numbers as it focuses on "concrete representation of the semantics" [71]. For this reason, it is not possible to represent register/memory values abstractly using GRIFT (i.e. not as natural numbers but, for example, as SMT expressions). To the best of our knowledge, our formal RISC-V model is the first executable model which focuses specifically on flexibility and thereby enables non-concrete execution of RISC-V instructions.

4.1.5.2. Modular Interpreters

Early work on modular interpreters for imperative languages [116] used monad transformers to compose the monads used to interpret the imperative features in a modular way. Monad transformers can be thought of as monads with a hole; instead of a monad `m` modeling a feature f (say, stateful computation), we give a monad transformer `m'` modeling the addition of feature f to an existing monad. This allows us to combine features in a "stack" of monads, and is implemented in Haskell in the `mtl`[11] library.

However, this approach has three drawbacks: firstly, the monad transformer already specifies the interaction with the other monad, so the approach is not truly compositional; secondly, it is not truly extensible, as once the monad stack is composed, no more monads can be added (this would result in a new monad stack); and thirdly, there is a severe performance cost for larger monad stacks [105, Section 4]. For these reasons, we use free monads with extensible effects which do not suffer from these drawbacks, even though lowering the performance penalty of free monads (cf. Subsection 4.1.4) is still an open challenge.

Our work is intended as a framework for abstract interpretation on machine code. Leveraging monads for this purpose, it is related to work on monadic abstract interpreters [165]. Besides the use of monad transformers in that work, there is one crucial difference: for software, abstract interpretation techniques extract the control flow graph

[11] `https://hackage.haskell.org/package/mtl`

of the program. Prior work by Sergey et al. uses continuation-passing style semantics for this [165]. We model the control flow implicitly using lazy evaluation; the next instruction is only fetched and decoded once it is needed.

4.1.5.3. Binary Software Analysis

Due to the utilization of free monads, we believe our RISC-V ISA model to be a versatile tool for implementing dynamic software analysis techniques that operate directly on the machine code level. Prior work has already demonstrated that it is feasible to implement techniques such as symbolic execution [84] or dynamic information flow tracking [142] for RISC-V machine code. However, this prior work does not leverage functional ISA specifications and thus relies on manual modifications of existing interpreters and is not easily applicable to additional RISC-V extensions or other ISAs (ARM, MIPS, etc.). For this reason, the majority of existing work on binary software analysis does not operate directly on the machine code level and instead leverages intermediate languages and lifts machine code to these languages [29, 41, 169].

This prior work therefore operates on a higher abstraction level and can thus not reason about architecture-specific details (e.g. instruction clock cycles) during the analysis. By building dynamic software analysis tools on an abstract ISA model, we can bridge the gap between the two approaches; we can operate directly on the machine code level while still making it easy to extend the analysis to additional instructions or architectures. This is especially important for modular ISAs like RISC-V.

4.1.6. Discussion and Future Work

So far, we have only applied our approach to the RISC-V architecture. Nonetheless, we believe the concepts described in Subsection 4.1.2.2 to be applicable to other architectures as well. We have, focused on the RISC-V architecture due to its simplicity as we consider the main contribution of this section to be the implementation of custom interpreters on top of a formal ISA model. A possible direction for future work would be focusing more on modeling aspects by supporting additional RISC-V extensions (especially from the privileged specification [154]), further RISC-V variants (e.g. 64- and 128-bit RISC-V), and maybe even additional ISAs (e.g. ARM). Alternatively, it would also be possible to perform further experiments with additional interpreters for our abstract ISA model. We are specifically interested in complementing our prior work on SymEx-VP, as presented in Chapter 3, with the formal ISA model proposed here. This integration should ease extending SymEx-VP to additional RISC-V extensions or even additional architectures. More broadly, one end goal of our work in this regard would be facilitating formal ISA models for the implementation of binary software analysis tools along the lines sketched in Subsection 4.1.5.3. Advances in this regard are presented in Section 4.2 and Section 4.3. Compared to the prevailing prior work on binary software analysis tools—which lifts machine code to an intermediate representation—we believe that building these tools on top of a formal ISA model also allows easier proofs of their correctness. An interesting direction for future work would therefore be investigating the issue of correctness of custom ISA interpreters. As illustrated in Figure 4.1, correctness proofs are paramount as we need to ensure that both the abstract and the actual semantics correctly implement the behavior mandated by the modeled ISA. Considering that our approach is specifically designed to support multiple actual semantics—through custom interpreters—manual validation is infeasible. Instead, it may be possible to leverage existing proof-assistant definitions for ISAs [5] to prove the correctness of created ISA interpreters through computer-aided theorem proving.

4.1.7. Conclusion

We have presented a flexible approach for creating functional formal models of instruction set architectures. The functional paradigm gives a natural and concise way to model the instruction format on different levels of abstraction, and the structuring mechanisms allow us to relate these levels. This way, by leveraging free monads, our approach separates instruction semantics from instruction execution. Contrary to prior work, our approach does not make any assumptions about the representation of memory/register values. Therefore, it can be used to implement software analysis techniques such as dynamic information flow tracking or symbolic execution; achieving the benefits outlined in Section 4.1.

We have demonstrated our approach by creating an abstract formal model of the RISC-V architecture. Based on this formal RISC-V model, we have created a concrete interpreter—which passes the official RISC-V ISA tests—for the 32-bit base instruction set and a custom interpreter for information flow tracking as a case study. An evaluation conducted with the Embench benchmark suite indicates that our concrete interpreter is faster than prior executable Haskell models of the RISC-V architecture. In future work, we would like to model additional extensions of the RISC-V architecture, perform further experiments with additional interpreters for our model, and investigate correctness proofs for these interpreters through computer-aided theorem proving. To stimulate further research in this direction, we have released our formal RISC-V model as open source software on GitHub.

4.2. Binary Symbolic Execution using Formal Semantics

In this section, we present a new symbolic execution engine which is based on the formal LibRISCV ISA model described in the previous section. This execution engine improves upon our prior work on the creation of symbolic ISSs as presented in Chapter 3. Instead of specifying the symbolic semantics manually for each instruction (see Subsection 3.1.2.3), we build upon the language primitives provided by LibRISCV and use them as an abstraction layer. This is similar to existing work which transforms the code to be tested to an IR, such as LLVM IR, and then symbolically executes this IR [49, 54, 41]. In this case, the IR acts as an abstraction layer for the formulation of symbolic instruction semantics. For a given piece of software in binary form, the IR is obtained through a *lifter* which transforms binary code instructions to the IR abstraction [144]. As such, symbolic semantics do not need to be specified manually for every instruction mandated by the targeted ISA but instead only for the more general IR abstractions. Furthermore, operating on a higher abstraction level—through an IR—also eases retargeting the analysis to different architectures, as the IR itself is architecture-independent. However, this established approach results in unique challenges when symbolically executing software for embedded devices. Contrary to software for conventional devices (laptops, desktops, and servers), embedded software interacts with the hardware on a low abstraction level through highly architecture-specific instructions, e.g. to configure interrupt handlers. These instructions are essential for the functioning of embedded software as they are used to implement context switching or dispatching of event handlers (see Section 1.1). As these instructions and their impact on the state of the underlying hardware platform are highly architecture-specific, they cannot be represented in an architecture-independent IR. For this reason, early work on symbolic execution of such software (e.g. Fie [54]) did not support code using such instructions. More recent work on Inception [49] attempts to mitigate this issue through a custom architecture-dependent lifting process and essentially models the low-level hardware state (registers, stack pointer, etc.) within the IR [49, Section 2.2]. Inception thereby sacrifices a major benefit of the IR-based symbolic execution approach—architecture independence—and is subject to errors and inaccuracies in the IR-based ad hoc implementation of the hardware state.[12] The symbolic ISS, presented in Chapter 3, was not subject to these limitations as it accurately executes binary code directly, but—contrary to FIE and Inception—it is challenging to extend it to additional instructions.

We aim to bridge the gap between IR-based symbolic execution approaches and our own aforementioned prior work. That is, we want to operate directly on binary code instructions (thereby supporting software embedded devices) while still making it possible to easily extend the analysis to additional instructions and thereby reduce manual effort. For this purpose, we contribute a novel symbolic execution approach which utilizes the language primitives of our LibRISCV ISA model as an abstraction layer. Because of the faithful representation of the ISA semantics, we can directly relate the symbolic execution to the binary code while still making it easy to extend it to additional instructions through the utilization of the language primitives. In the following, we present a prototype implementation of this approach called BinSym and demonstrate its extensibility by conducting a case study with the RISC-V M-extension [153, Chapter 7]. Furthermore, we conduct an experimental comparison with IR-based symbolic execution approaches proposed in prior work. As part of the experiments, we uncovered several previously unknown inaccuracies in the RISC-V lifter provided by prior work on angr [169], thereby illustrating that lifter-induced programming errors and inaccuracies are by no means an academic issue.

4.2.1. ISA Semantics for Symbolic Execution

In the following, we present our approach and make a case for using formal ISA semantics as a building block for the implementation of symbolic execution engines for binary code. Furthermore, we introduce BinSym, a prototype implementation of our approach which symbolically executes RISC-V binary code.

4.2.1.1. Formal Semantics for Binary Code

While early works on symbolic software execution required access to the source code [32, 162, 31], it is nowadays becoming increasingly popular to operate directly on binary code [169, 61, 41]. Doing so enables program analysis through symbolic execution, even if the source code is not available. Furthermore, it ensures that the same code is being tested that is later deployed in a production environment, thereby enabling reliable reasoning about low-level details such as memory use or timing, which are particularly

[12] Any inaccuracy in the lifter may lead to bugs being missed in the tested software [88, Section 2.2].

important in the embedded domain. In order to track performed computations and to formally reason about branch points in a program, executing binary code symbolically requires implementing symbolic semantics for the binary code instructions provided by the targeted ISA. Since mainstream ISAs mandate hundreds of instructions, this can be a cumbersome and error-prone task, especially considering that the symbolic semantics need to conform to the ISA specification. Otherwise, bugs may be missed in the tested software. In order to simplify the implementation of symbolic semantics, existing work does not operate directly on ISA instructions but instead uses IRs (e.g. DBA [13] or Valgrind VEX [133]) as an abstraction [144]. As illustrated by the gray arrows on the left-hand side of Figure 4.3, binary code for a given ISA is transformed to this IR using a lifter. The lifter receives binary code as an input and produces IR as an output, its implementation is therefore ISA-specific. Conceptually, the lifter thus implicitly provides an ad hoc implementation of the ISA semantics. This implementation is manually written and not based on a formal model of instruction semantics. Therefore, it is—as we will see in Subsection 4.2.2.2—subject to errors. Since the IR is architecture-independent, behavior that cannot be expressed in an architecture-independent way (e.g. interrupt handling) is lost during the lifting process [49, Section 2.2]. This prevents the symbolic execution engine, which implements symbolic semantics on the IR abstraction, from reasoning about architecture-specific behavior, which is paramount when testing software for embedded devices.

Fundamentally, the issue with the prevailing IR-based approach is that ISA semantics and symbolic semantics are specified on different abstraction levels. Therefore, the symbolic execution cannot be directly related to the binary code. We attempt to overcome this issue by formulating both the ISA semantics and the symbolic semantics on the same abstraction level. As illustrated by the black path on the right of Figure 4.3, we achieve this by leveraging existing formal models of ISA semantics [5, 26, 161]. These formal models provide us with an accurate machine-readable specification that is faithful to the ISA and its mandated instruction semantics. Conceptually, formal models describe instruction semantics in formal—instead of natural—language using language primitives. Instead of operating on an IR, our approach uses these language primitives as an abstraction for both the description of ISA semantics and the description of symbolic semantics. Since both semantics are specified on the same abstraction level, we can symbolically reason about low-level ISA details such as register file accesses or program counter changes for interrupt handling. At the same time, we do not need to manually specify symbolic semantics for every ISA instruction, which would be cumbersome for

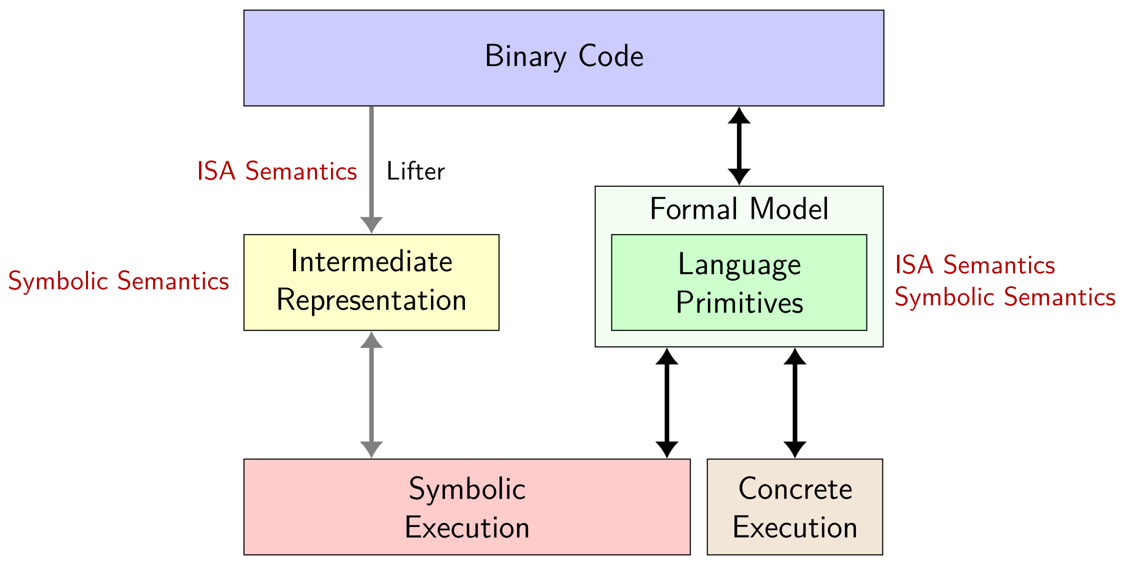


Figure 4.3.: Comparison of IR-based symbolic execution (left) and our approach (right). In the existing IR-based approach, we cannot refer back to the binary code from the symbolic execution engine because the ISA semantics are encoded implicitly in the lifter. In our approach, because of the faithful representation of the semantics with a formal model, both symbolic and concrete execution can be related to the binary code.

ISAs with hundreds of instructions. Instead, we only need to specify symbolic semantics for a handful of language primitives as the instructions are formally described in terms of these primitives. This enables our approach to accurately operate on the instruction-level while still allowing the analysis to be easily extended to additional instructions or even architectures.

Prior work on Sail [5] has demonstrated that it is feasible to describe different architectures (ARM, RISC-V, MIPS, IBM Power, and x86) using the same language primitives. Related work has also already used formal models of instruction semantics for concrete software execution [5, 26, 161], but our work is—to the best of our knowledge—the first which proposes using them for binary analysis through DSE. Building both symbolic and concrete execution on the same abstraction also allows for their composition and the reuse of existing components. In the following, we demonstrate this property by presenting a prototype implementation of our outlined symbolic execution approach.

4.2.1.2. Formal Symbolic Execution of RISC-V Binary Code

In order to evaluate our proposed approach, we have created a prototype implementation that we refer to as BinSym. BinSym symbolically executes binary code for the open standard RISC-V [153] architecture. We chose RISC-V for our work because, due to its openness, it has enabled a large body of research on formal ISA specifications. Furthermore, RISC-V is a modular architecture, i.e. it consists of a base instruction set and optional extensions, which are implemented on top and can be combined as needed. Therefore, it benefits from an extensible symbolic execution approach as the specification is constantly expanding, requiring binary analysis tools to "catch up" by implementing new extensions. Without formal specifications, this is a manual, error-prone process.

As discussed previously, a variety of different formal ISA specifications for the RISC-V architecture have emerged in recent years [152]. Naturally, for our BinSym implementation, we build upon the LibRISCV formal specification presented in Section 4.1, as it was specifically designed for an application of symbolic execution. In Section 4.1, LibRISCV has already been used for concrete execution of 32-bit RISC-V machine code; parts of this concrete interpreter can be reused in BinSym. A comprehensive discussion of existing work on formal ISA semantics, in the context of binary analysis, is provided in Subsection 4.2.3.

Language Primitives The premise of our proposed symbolic execution approach is to use the primitives of a formal ISA model as an abstraction layer. In the following, we introduce the language primitives that LibRISCV provides to formally describe the semantics of RISC-V instructions. As an example, we discuss the formal description of the `BEQ` (branch-if-equal) instruction. We chose a branch instruction here since these instructions are central to symbolic execution for reasoning about branch points in a program. The `BEQ` instruction is formally described in LibRISCV as follows:[13]

```
instrSemantics BEQ = do
  rs1 <- decodeRS1 >>= readRegister
  rs2 <- decodeRS2 >>= readRegister
  imm <- decodeImmB

  runIfTrue (rs1 `Eq` rs2) $ do
    writePC (pc `Add` imm)
```

[13]For clarity, the description has been slightly simplified.

The formal description starts off by specifying the operands of the instruction (Line 2 - Line 4). The RISC-V `BEQ` instruction has two register operands and one immediate operand, which are encoded in the instruction and extracted in Line 2, Line 3, and Line 4. The register operands specify the values that should be compared for equality, while the immediate operand specifies the amount by which the program counter (PC) is incremented if they are equal [153, Section 2.5]. The comparison and increment of the program counter are performed in Line 6 - Line 7. Conceptually, the instruction is formally described in terms of two kinds of primitives: stateful constructs (`decode`, `readRegister`, `runIfTrue`, and `writePC`) as well as arithmetic and logic expressions (`Eq` and `Add`). In LibRISCV, the former are modeled using a monad while the latter is just a polymorphic algebraic data type. In the following, we provide more details on these primitives and describe how we use them for symbolic execution.

Expression Abstraction In order to build a symbolic execution tool as a custom ISA interpreter, we need to evaluate arithmetic and logic operations as SMT bit-vector expressions. For this purpose, we use existing Haskell bindings for the popular SMT solver Z3 [125] and map constructs of the existing LibRISCV bit-vector expression language to SMT bit-vector operations using pattern matching:

```
evalE :: Z3.MonadZ3 z3 => E.Expr Z3.AST -> z3 Z3.AST
evalE (E.FromInt n v) = Z3.mkBitvector n v
evalE (E.ZExt n v)     = evalE v >>= Z3.mkZeroExt n
evalE (E.SExt n v)     = evalE v >>= Z3.mkSignExt n
evalE (E.Add e1 e2)    = binOp e1 e2 Z3.mkBvadd
evalE (E.Sub e1 e2)    = binOp e1 e2 Z3.mkBvsub
evalE (E.Eq e1 e2)     = binOp e1 e2 Z3.mkEq     >>= fromBool
evalE (E.Slt e1 e2)    = binOp e1 e2 Z3.mkBvslt  >>= fromBool
--- ...
```

However, we do not want to use SMT expressions for all register and memory values during symbolic execution of binary code. When executing binary code, there is no clear distinction between code and data. Conceptually, the next instruction (i.e. code) is also just a fixed-width value loaded from memory. Representing instructions as SMT values would result in a significant performance penalty as they would need to be converted to a concrete value—on each instruction fetch—in order to be decoded, an expensive operation referred to as concretization. To overcome this limitation, we use concolic execution in our prototype implementation. As explained in Section 2.3, concolic execution allows for efficient concretization for instruction decoding. This is achieved by executing the program with concolic values: tuples with a concrete and an optional symbolic part. Arithmetic and logic operations from the LibRISCV expression language are then performed on both parts. In order to implement concolic execution, we therefore compose the `evalE` function shown above (which performs symbolic expression evaluation) with the existing LibRISCV expression evaluator for concrete execution. The resulting function performs concolic expression evaluation.

Stateful Constructs In addition to the expression abstraction, we also need to provide symbolic semantics for the stateful constructs of LibRISCV. As we have seen in the example `BEQ` description, these constructs are used to interact with the hardware state (e.g. the register file or memory). In LibRISCV, they are specified using a Generalized Algebraic Data Type (GADT) [208]. The resulting data type is called `Operations` and is defined as follows:

```
1 data Operations v r where
2   WriteRegister :: v -> v -> Operations v ()
3   ReadRegister :: v -> Operations v v
4   StoreMem :: ByteSize -> v -> v -> Operations v ()
5   LoadMem :: ByteSize -> v -> Operations v v
6   RunIf :: E.Expr v -> Operations v () -> Operations v ()
7   -- Additional LibRISCV Operations constructors omitted for clarity.
```

The `Operations` type is parameterized over a generic type `v` which is instantiated with a type representing the values on which the constructs operate (i.e. concolic values in our case) and a type `r` which specifies the return type of the operation. Based on these types, `Operations` defines constructors used to describe interactions with the hardware state. For example, `WriteRegister` (Line 2) takes a register index as well as a value that should be written to the register (both represented as a generic type `v`) and returns an `Operations` instance parameterized over `v` that has no meaningful return type, i.e. returns the unit type `()`. As illustrated in the prior `BEQ` example, these constructors are used within the formal instruction description. Recall that LibRISCV composes constructors through the use of free monads [105], thus allowing for compositional semantics (see Subsection 4.1.2.2). We achieve concolic execution through an interpretation of the free monad, i.e. the composed constructors of the `Operations` data type. For this purpose, we implemented custom variants of the register file and the memory that are capable of operating on concolic values. We were able to partially reuse existing implementations from LibRISCV for this purpose. For example, LibRISCV offers a polymorphic implementation of a register file which is parameterized over a value type. We then map constructors such as `ReadRegister`, `WriteRegister`, `LoadMem`, or `StoreMem` to the custom implementation of our register file and memory. This enables us to execute RISC-V instructions with operands representing concolic values. As explained in Section 2.3, we can then use these operands to symbolically reason about branches in the program. For this purpose, we need to implement custom handling of branch conditions. Such branch conditions are denoted via the `RunIf` constructor from the `Operations` data type (Line 6), we have already seen it in use through the `runIfTrue` abstraction in the formal `BEQ` description. For concolic execution, we interpret the constructor as follows: we take the branch according to the concrete value, but if the concolic value has a symbolic part, then we track the branch condition in an execution trace. From this execution trace, we iteratively build an execution tree, where each node is a branch point in the program with a condition that depends—directly or indirectly—on a symbolic value. This binary tree is then used to implement a dynamic symbolic execution algorithm, which iteratively restarts execution with new input values, as outlined in Section 2.3.

Summary We have presented BinSym, an exemplary prototype implementation of our accurate and extensible symbolic execution approach for the RISC-V architecture. BinSym is written in roughly 1000 LOC in Haskell and implements standard dynamic symbolic execution [10, Section 2.1] with random path selection [10, Section 2.2] and an address concretization memory model [10, Section 3.2]. The implementation is freely available on GitHub.[14]

4.2.2. Evaluation

In the following, we evaluate our approach using the presented BinSym symbolic execution engine. Regarding the evaluation, we are interested in the following research questions: (a) can the symbolic analysis be easily extended to support additional instructions? (b) does instruction-level symbolic execution offer competitive performance in comparison with IR-based symbolic execution?

[14]https://github.com/agra-uni-bremen/binsym

4.2.2.1. Extensibility Case Study

A major advantage of IR-based symbolic execution approaches is that they can be easily extended to additional instruction set extensions or even architectures, as doing so only requires changes to the lifter and not to the symbolic execution engine itself. In the following, we demonstrate that we can achieve the same property in our approach using the language primitives of a formal ISA model; the key here is the compositionality of the LibRISCV semantics. For this purpose, we conduct a case study with our BinSym symbolic execution engine and the underlying LibRISCV formal model used by our prototype implementation. In order to support additional instructions in our prototype, we first need to express these instructions in the formal language used by LibRISCV. As long as these additional instructions can be expressed in terms of the existing language primitives, no changes are necessary to BinSym. However, some instructions may require modifications of the language primitives, for example, new operations in the LibRISCV expression abstraction. In this case, BinSym will also need to be modified to provide symbolic semantics for these operations. Fortunately, the modular design of our prototype implementation makes it easy to perform such modifications. In the following, we conduct a case study with the standardized RISC-V instruction set extension for integer multiplication and division (the M-extension) to demonstrate this property.

The M-extension specifies eight additional RISC-V instructions for multiplication and division [153, Section 7]. In order to formally describe these instructions, we added five additional constructors to the expression abstraction:

```haskell
data Expr a =
  -- Existing LibRISCV constructors omitted for clarity.
  Mul  (Expr a) (Expr a) |
  UDiv (Expr a) (Expr a) |
  SDiv (Expr a) (Expr a) |
  URem (Expr a) (Expr a) |
  SRem (Expr a) (Expr a)
```

The added constructors are binary operations which perform the following arithmetic operations: multiplication, unsigned/signed division, and unsigned/signed division with remainder. Furthermore, we added support for these new constructors to the existing LibRISCV interpreter for concrete execution. As explained in Subsection 4.2.1.2, this is necessary as we reuse this interpreter for concolic execution. The concrete interpreter also passes the official RISC-V ISA for the M-extension, which indicates that it has been modeled correctly.[15] Based on the added multiplication and division operations, we were able to formally describe the eight instructions of the M-extension using the LibRISCV language primitives. The formal description of all eight instructions requires 60 lines of Haskell code and is available as part of the publication artifacts. As an example, the formal description of the `DIVU` instruction—which performs division of unsigned integers—looks as follows [153, Section 7.2]:

```haskell
1 instrSemantics DIVU = do
2     (r1, r2, rd) <- decodeAndReadRType
3     let cond = fromImm r2 `Eq` fromInt 0
4     runIfElse cond
5         do $ WriteRegister rd (fromInt 0xFFFFFFFF)
6         do $ WriteRegister rd (r1 `UDiv` r2)
```

The `DIVU` instruction interprets the register operands as unsigned integers and divides them using the new `UDiv` operation (Line 6). As per the RISC-V specification, the instruction also has special handling for division by zero and returns $2^{32} - 1$ in this case (i.e. `0xFFFFFFFF`, see Line 5) [153, Table 7.1]. Naturally, this case distinction within the instruction is also included in the symbolic reasoning performed by the BinSym symbolic execution engine (e.g. if the divisor is an unconstrained symbolic value). In order to support the formally described eight additional instructions from the M-extension in BinSym, we needed to add symbolic semantics for the new division and multiplication operations. As explained in Subsection 4.2.1.2, this is achieved by mapping them to Z3 expressions:

```haskell
evalExpr :: Z3.MonadZ3 z3 => E.Expr Z3.AST -> z3 Z3.AST
-- ...
evalExpr (E.Mul e1 e2)  = binOp e1 e2 Z3.mkBvmul
evalExpr (E.SDiv e1 e2) = binOp e1 e2 Z3.mkBvsdiv
evalExpr (E.UDiv e1 e2) = binOp e1 e2 Z3.mkBvudiv
evalExpr (E.SRem e1 e2) = binOp e1 e2 Z3.mkBvsrem
evalExpr (E.URem e1 e2) = binOp e1 e2 Z3.mkBvurem
-- ...
```

No additional changes are necessary in BinSym to support the M-extension as otherwise, the formal description for these new instructions only relies on existing LibRISCV language primitives.

[15] The RISC-V tests provide extensive unit tests for each instruction, including common edge cases. Refer to `https://github.com/riscv/riscv-tests` for more information.

Note that the outlined changes to the LibRISCV expression abstraction were only necessary as the original LibRISCV version—presented in Section 4.1—only supported the RISC-V base instruction set. We envision that LibRISCV will support all common RISC-V extensions in the future and prior work on similar formal ISA models has demonstrated that this is feasible [5, 26, 161]. Nonetheless, the ability to extend the symbolic analysis to additional instructions is an important property to also support custom domain-specific instructions, support for which is particularly relevant when analyzing software for embedded systems [51, Section 2]. In this context, the case study serves to demonstrate that our approach can be easily extended to support additional instructions. In the next section, we evaluate the performance of our prototype implementation and also use instructions of the M-extension for this purpose.

4.2.2.2. Performance Benchmarks

Given the large state space of real-word applications, good execution performance is a vital property of any symbolic execution engine. In the symbolic execution domain, we can distinguish concrete and symbolic execution performance. Symbolic execution is only performed when the executed code interacts with symbolic variables. As explained in Subsection 4.2.1.2, we reuse the concrete interpreter from LibRISCV for concrete instruction execution in our BinSym symbolic execution engine. The performance of this concrete interpreter has already been evaluated in Subsection 4.1.4. Therefore, we focus on symbolic execution performance in this section. For this purpose, we perform experiments with synthetic benchmarks that interact heavily with symbolic variables and compare the execution time with existing symbolic execution engines.

Since BinSym executes 32-bit RISC-V binary code symbolically, we only conduct a comparison with prior work that also supports RISC-V. Apart from our own work on SymEx-VP (Chapter 3), we are presently aware of the following engines which fulfill this criterion: angr [169] which lifts RISC-V binary code to Valgrind VEX [133] and BinSec [61] which lifts binary code to DBA [13].[16] Comparing these engines is challenging; they implement different symbolic execution algorithms and target different kinds of software. For example, SymEx-VP is explicitly designed to test software for embedded devices, while BinSec was primarily designed for and evaluated on Unix programs (i.e. software for non-embedded devices). Additionally, BinSec implements static symbolic execution while SymEx-VP and BinSym implement DSE [10]. In order to compare

[16]Technically, BAP [29] also supports RISC-V binary code but the public open source release was insufficiently documented for us to get its symbolic execution plugin to work.

Table 4.2.: Average execution time as an arithmetic mean for five synthetic benchmark applications, each executed with four symbolic execution engines five times respectively. The last column specifies the standard derivation. Engines marked with a † symbol did not discover all execution paths on the given benchmark (compare the #Paths column).

Benchmark	Engine	#Paths	Mean Time	Derivation
`base64-encode`	BinSym	6250	169.7 s	0.88 s
	SymEx-VP	6250	217.5 s	1.81 s
	BinSec	6250	229.2 s	2.27 s
	angr †	125	32.8 s	0.41 s
`bubble-sort`	BinSym	720	52.1 s	0.03 s
	SymEx-VP	720	44.2 s	0.42 s
	BinSec	720	82.3 s	0.44 s
	angr	720	256.6 s	1.16 s
`is-prime`	BinSym	101	98.2 s	0.44 s
	SymEx-VP	101	129.5 s	0.53 s
	BinSec	101	135.7 s	1.52 s
	angr	101	207.3 s	1.92 s
`insertion-sort`	BinSym	5040	67.0 s	0.02 s
	SymEx-VP	5040	122.3 s	0.83 s
	BinSec	5040	128.4 s	0.90 s
	angr	5040	392.8 s	1.64 s
`uri-parser`	BinSym	1390	53.8 s	0.41 s
	SymEx-VP	1390	67.2 s	0.43 s
	BinSec	1390	94.5 s	0.54 s
	angr †	1386	321.9 s	1.76 s

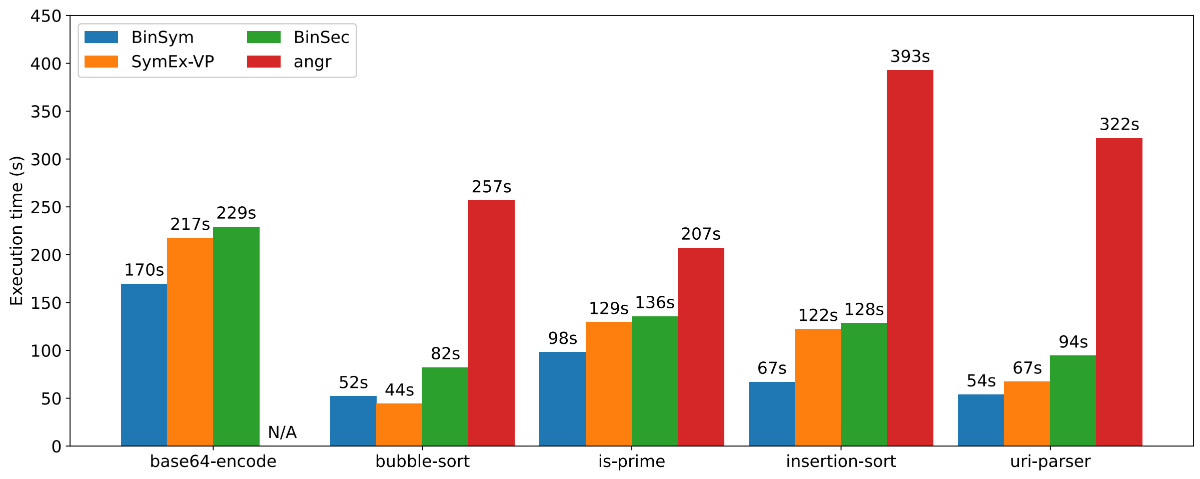


Figure 4.4.: Visualization of the raw data from Table 4.2, i.e. average execution time as an arithmetic mean over five executions per benchmark *(lower is better)*.

these different tools, we came up with a set of synthetic portable benchmarks that are operating system independent (e.g. sorting algorithms or parsers). Similar benchmark applications have been used in prior work for evaluation purposes [49, Section 4.2]. Since we focus on symbolic execution performance, these benchmarks operate on a fixed-size input of symbolic variables. The size of the symbolic input has been adjusted across all benchmarks to yield similar execution time results. Furthermore, to account for differences regarding implemented symbolic execution algorithms, special care has been taken to ensure that all benchmarks are fully explorable within a reasonable time span, i.e. that all engines can discover the same amount of execution paths on all benchmarks.[17] Similarly, the benchmark applications also do not access memory with symbolic addresses, thereby avoiding that different memory models affect the evaluation results [25]. Lastly, all tested symbolic execution engines have been configured to use Z3 [125] as an SMT solver to increase comparability and avoid benchmarking the underlying solvers of these engines.

In total, we executed five synthetic benchmark applications as 32-bit RISC-V binary code with four symbolic execution engines. Each benchmark application has been executed five times with each engine in a Docker container on an Intel Xeon Gold 6240 system running Ubuntu 22.04. The arithmetic mean over all five executions as well as the standard derivation per benchmark and execution engine are given in Table 4.2.

[17]Otherwise, some symbolic execution engines may discover more time-consuming paths that are not explored by other engines, thus distorting the benchmark results.

Furthermore, the results are visualized as a grouped bar chart in Figure 4.4. In Figure 4.4, the absolute execution time—as the arithmetic mean over all five executions—is given in seconds on the y-axis while the x-axis lists the benchmark applications. For each benchmark application, four bar charts are given which correspond to the aforementioned symbolic execution engines; from left to right: BinSym (blue), SymEx-VP (yellow), BinSec (green), and angr (red). For the `base64-encode` benchmark, no results are given for angr as it failed to discover 6125 execution paths (see Table 4.2). These execution paths were found by all other tested engines. This is a programming error in angr's RISC-V lifter, which may cause angr to miss bugs in tested software and illustrates the importance of implementing the ISA semantics correctly. Similarly, angr also misses four paths on the `uri-parser`, but these four paths do not impact the execution time results in Figure 4.4 significantly. Nonetheless, the `uri-parser` benchmark results also affirm that there are bugs in angr's lifter for the RISC-V architecture. We have further investigated and reported these bugs to angr developers.[18] The bugs and the benchmark results can be reproduced using the paper artifacts [186].

Regarding the interpretation of the results, the standard derivation (given in Table 4.2) is negligible and execution time results are consistent across all benchmark applications: angr is by far the slowest engine, BinSec, SymEx-VP, and BinSym have similar execution time results, with BinSym being faster than BinSec and SymEx-VP on most benchmarks. An exception in this regard is the `bubble-sort` benchmark, where SymEx-VP is slightly faster than BinSym. We attribute this to the fact that this benchmark is solver-intensive and—contrary to BinSym—SymEx-VP employs additional SMT solver optimizations on top of Z3 that have been pioneered by KLEE (e.g. a counterexample cache), which seem to be beneficial for this particular benchmark [32, Section 3.3]. SymEx-VP is slower than BinSym on all other benchmarks because it executes software in a SystemC [180] simulation environment. SystemC is a C++ library for modeling hardware peripherals, it is used by SymEx-VP to support execution of embedded firmware, which interacts closely with such peripherals but comes with a significant performance penalty (see Section 3.1). BinSym does not support hardware peripheral models and is therefore faster. Similarly, angr is the slowest symbolic execution engine as it is the only engine which is written in an interpreted scripting language (Python), while all other engines are written in compiled languages (C/C++, OCaml, and Haskell). As prior work pointed out, this makes angr slower but easier to modify—similar to the prior SystemC remark—a not to be underestimated aspect which is difficult to account for

[18] `https://github.com/angr/angr-platforms/pull/64`

in an empirical performance evaluation [144, Section 6.4]. Furthermore, BinSec spawns a new Z3 process for each query when configured to use the Z3 solver while BinSym, SymEx-VP, and angr use the API provided by the Z3 library. Since the benchmarks require solving a lot of queries, as they focus on evaluating symbolic execution performance, process forking overhead can contribute significantly to the overall execution time.

In conclusion, the benchmarks nonetheless demonstrate that the well-known performance costs associated with free monads [105, Section 2.6] are not a bottleneck for the implementation of a symbolic execution engine, as BinSym offers competitive execution time performance compared to existing symbolic execution engines for binary code. Especially in comparison to our own prior work on SymEx-VP, which implements the same symbolic execution algorithms without the utilization of formal ISA semantics. Comparing these results with additional prior work is difficult as different design decisions contribute to the performance of a symbolic execution engine and isolating them is challenging [144, Section 6.2]. Most importantly, the comparison with BinSec is limited as it only provides preliminary support for the Z3 solver, forking a new Z3 process for each query and thereby incurring a significant performance penalty. Furthermore, we also uncovered bugs in angr's manually-written lifter for the RISC-V architecture. These bugs can cause angr to miss programming errors in real-world software and illustrate the importance of implementing the ISA correctly. By leveraging formal models of instruction semantics, we make a significant contribution towards ensuring the correctness of symbolic execution engines with regard to the ISA specification.

4.2.3. Related Work

There is a large body of related work on symbolic execution [10]. Early works in this domain required access to the source code [32, 162, 31]. The most popular source-based symbolic execution tool is KLEE [32]. KLEE requires the source code to be translated to LLVM IR using the LLVM compiler infrastructure and then symbolically executes the IR. Initial work on symbolic execution of binary code builds upon KLEE and therefore transforms binary code to LLVM IR through an ISA-specific lifting process [41, 54, 49]. In recent years, additional tools have emerged that use other IRs such as Valgrind VEX [133, 169], DBA [13, 61], or even custom ones [29]. As discussed in Section 4.2, it is challenging to support ISA-specific instructions in such approaches. Unfortunately, these instructions are commonly used in embedded software and without supporting them such software cannot be symbolically executed. Prior work on Inception acknowledges this problem and attempts to mitigate it by modeling the architecture state (register file, pending interrupts, et cetera) within the lifted IR [49]. Contrary to our work, this model is manually written and not based on a formal specification; thus, its implementation is error-prone and not easily extensible. A comparison of source-based and lifting-based approaches is provided by Poeplau et al. [144]. As an alternative to binary lifting, prior work on QSYM has achieved symbolic execution through binary instrumentation and tightly couples it with native instruction-level execution [209]. However, this work is not applicable to the embedded domain, as native execution on a constrained embedded device is significantly slower than emulation. Similar to BINSYM our own work on SYMEX-VP also operates directly on binary code and symbolically executes RISC-V binary code, however, SYMEX-VP manually specifies symbolic semantics for each RISC-V instruction and is therefore not extensible. Furthermore, as the symbolic semantics are specified manually, there is a large margin for error, which we mitigate in our work through machine-readable formal models of instruction semantics.

Prior work has presented different approaches for providing formal ISA semantics [5, 161, 26]. The most comprehensive work in this domain is Sail which provides formal semantics for ARM, RISC-V, MIPS, IBM Power, and x86 using a custom DSL [5]. Contrary to other prior work, Sail is not directly executable; the DSL needs to be translated to C or OCaml in order to achieve binary code execution. As execution is the focus of our work, we build upon an executable formal model which is written in a general-purpose programming language. Several executable formal models have been presented in related work [26, 161]. We made use of work on LIBRISCV (presented in Section 4.1) for the prototype implementation of our symbolic execution approach, as it is specifically tailored to the creation of custom ISA interpreters in the lineage of prior work on modular interpreters [116, 115, 177]. Modular interpreters specify program semantics in a composable and modular way and therefore ease the creation of custom program interpreters. Related work on TSL has applied these concepts to the creation of abstract interpreters [117]. Furthermore, Goel et al. also utilize formal semantics for automated proofing of properties on x86 binary code [73]. This approach is based on a partial, formal model of the x86 architecture [74]. It is closely related to our work as it also performs symbolic execution but focuses on completeness to prove properties about the executed code; therefore, its applicability to real programs is limited, e.g. efficient handling loops is posed as a challenge for future work [73, Section 6]. In contrast, our approach is specifically designed for program analysis utilizing DSE to improve scalability.

4.2.4. Discussion and Future Work

In future work, we would like to expand the underlying formal ISA model for the RISC-V architecture provided by LibRISCV. With our enhancements, LibRISCV supports the 32-bit base instruction set and the M-extension. We would be interested in modeling additional standardized extensions; prior work on similar executable Haskell models has demonstrated that this is feasible [26, 161]. In this context, it is also deemed worthwhile to investigate if such executable ISA models can be expanded to support additional architectures such as MIPS or ARM. Apart from such modeling aspects, we are also interested in correctness. That is, we would like to prove that the symbolic instruction semantics of our BinSym prototype implementation conform to the RISC-V specification. Doing so requires proofing the formal model of LibRISCV and the symbolic interpreter provided by BinSym for the formal model. For this purpose, we want to leverage existing theorem prover definitions for Coq and Isabelle provided by prior work for different ISAs [5]. Compared to prior work on IR-based symbolic execution, employment of formal ISA semantics eases performing such proofs.

4.2.5. Conclusion

We have presented a novel approach for symbolic execution of binary code that accurately operates on the instruction-level. In order to cope with the complexity of modern ISAs, we leverage formal models of instruction set semantics and thereby allow the analysis to be easily extended to additional instructions. By operating directly on binary code instructions, our approach can easily support architecture-specific interactions with the low-level hardware state (e.g. register file accesses for interrupt handling), which are common in the embedded domain. This is an improvement over prior work on IR-based symbolic execution which does not support such interactions (as it operates on a portable architecture-independent IR) and is therefore incapable of symbolically executing embedded software. In comparison to our own prior work on SymEx-VP, which also does not operate on an IR, the compositionality of the semantics of the underlying formal model makes our proposed approach easily support additional instruction set extensions, which is not possible with SymEx-VP. In future work, we plan to expand the comprehensiveness of the formal ISA model we used in our prototype implementation. Additionally, we want to investigate correctness proofs for both the formal ISA model and the symbolic semantics specified on top of it. As evident by our experiments with prior work on angr, correctness is vital in this domain, as otherwise bugs may be missed.

4.3. Generation of Instruction Set Simulators

In Section 4.2, we have presented a novel symbolic execution approach which derives a symbolic ISS from a formal ISA model. For this purpose, we built a custom ISA interpreter on top of a free monad construction in the Haskell programming language. Naturally, this ISA interpreter is also written in Haskell and therefore as-is incompatible with the SystemC environment modeling approach presented in Chapter 3 which is based on C/C++. In this section, we outline a path towards an integration of SystemC hardware models with our formal RISC-V ISA model. For this purpose, we pursue a complementary direction regarding the use of formal ISA models: code generation. That is, instead of building an ISS as a custom ISA interpreter directly in Haskell, we generate a C/C++ ISS from the LibRISCV Haskell description. Thereby achieving an integration of LibRISCV with VPs such as SymEx-VP or `riscv-vp`. This allows us to bridge the gap between our contribution in the environment modeling domain (Chapter 3) and our work on formal semantics for symbolic execution (Section 4.1 and Section 4.2). In this section, our contributions towards this goal are:

1. An improved version of the LibRISCV ISA model, originally presented in Section 4.1, which has been enhanced for the purpose of code generation.
2. A C/C++ code generator that generates a minimal ISS which can be easily integrated with different existing simulators.
3. A modified version of the popular Spike [200] and `riscv-vp` [85] simulators, which use a generated ISS (instead of a manually written one).

To the best of our knowledge, the ISS generation approach presented here is the first which is easily applicable to a variety of existing RISC-V simulators. Thereby easing the application of our proposed symbolic execution approach to additional VPs and simulators in general. The experiments we have conducted with Spike [200] and `riscv-vp` [85] confirm the feasibility of our approach for this purpose. Furthermore, performed benchmarks indicate that an ISS generated using our tooling achieves competitive simulation speed compared to a manually written one while still passing the official RISC-V ISA tests.

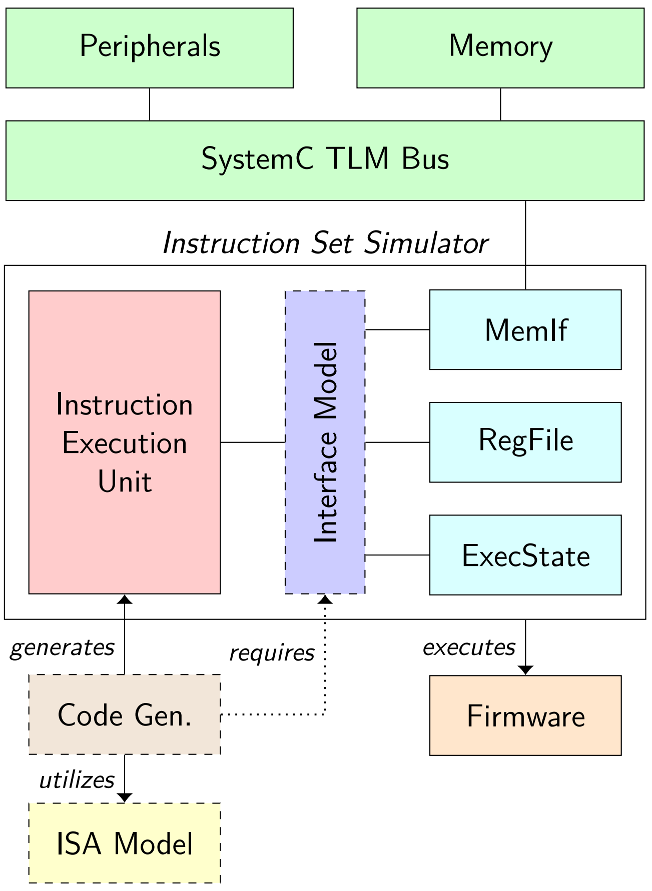


Figure 4.5.: Overview of our minimally invasive ISS generation approach.

4.3.1. Approach

In the following, we present our approach for minimally invasive generation of ISSs from formal ISA models. For this purpose, we first provide a high-level overview and then discuss different components of our approach in greater detail.

4.3.1.1. Overview

Figure 4.5 provides an overview of our approach and includes an illustration of the software architecture of an ISS with some VP-specific components (e.g. a SystemC TLM bus). Components added for the application of our approach are highlighted using a dashed box. The ISS in Figure 4.5 consists of different internal components and is responsible for executing a firmware image as faithful to a real processor as possible. Regarding the internal ISS components, we differentiate between the instruction execution unit (which is responsible for the execution part of the fetch-decode-execute cycle) and architectural state components (e.g. the register file) which are required for instruction execution but are conceptually separate components. We concentrate on the generation of the instruction execution unit, where the majority of modifications for an integration with symbolic execution occur (see Subsection 3.1.2.3 and Section 4.2). While we focus on concrete execution for now, we can prospectively ease the application of our proposed symbolic execution approach to additional VPs and simulators by generating the execution unit from a formal model. In order to generate this component, the code generation tool needs to be able to emit code that interacts with the architectural state components to yield code which implements the instruction semantics (e.g. to write a register). Since the API of these components is highly simulator- and vendor-specific—and we want to be able to support different existing simulators—we leverage a custom *interface model* for our approach. This interface model provides a generic API for common operations (e.g. writing/reading registers or accessing memory). The generic API itself is a set of C function prototypes which define a simulator-agnostic interface for performing these common operations (see Subsection 4.3.1.3). These functions need to be implemented manually once on a per-simulator basis by mapping them to the internal interfaces provided by the simulator. Since simulator-specific code is abstracted through the generic API, the code generation tool is itself applicable to different RISC-V simulators (see Subsection 4.3.2.1). While we believe the outlined approach to be practical for different ISAs, we focus on the RISC-V to allow for an integration with `riscv-vp` and—in future work—SymEx-VP. In the next subsection, we present enhancements of LibRISCV to allow its utilization for code generation purposes.

4.3.1.2. ISA Model

As discussed in Section 4.3, we are using the existing LibRISCV formal ISA model for our approach. The benefit of LibRISCV in the context of code generation is that—in contrast to prior work—it describes instructions semantics in isolation without providing a formal description of other ISA aspects such as memory behavior or decoding. This allows us to only generate the code implementing instruction semantics (the instruction execution unit) from the formal specification while retaining other parts as-is, thereby making our approach minimally invasive and easing the integration with existing simulators. Furthermore, for an implementation of symbolic execution, we primarily need to make modifications to the code implementing instruction semantics and do not need to change the decoder, for example.

As per Section 4.1, LibRISCV leverages a Haskell EDSL for the formal description

```
1 semantics LBInst{rd=dest, rs1=reg, imm=off} = do
2   base <- readRegister reg
3   byte <- loadByte (base `Add` off)
4   writeRegister dest (SExtByte byte)
```

Listing 4.8.: Simplified description of LB instruction semantics in LibRISCV.

of RISC-V instruction semantics. Recall that this EDSL consists of two components: (1) primitives for describing interactions with architectural state components (e.g. the memory) and (2) an expression language for performing operations on memory/register values. In order to illustrate the interaction with architectural state components, the formal description of the RISC-V `LB` instruction in this EDSL is provided in Listing 4.8. The semantics of this instruction are described in terms of the `readRegister` (Line 2), `loadByte` (Line 3), and `writeRegister` (Line 4) primitives which correspond to changes of the architectural state. Furthermore, operations on retrieved register/memory values are modeled using the aforementioned expression language, i.e. the `Add` and `SExtByte` constructors in Listing 4.8. For the purpose of code generation, we need to map constructors of the LibRISCV expression language to C/C++ expressions. Additionally, we need to map the `readRegister`, `writeRegister`, etc. primitives to functions provided by our interface model.[19]

In order to do so, we further enhanced the existing ISA model for code generation purposes. As discussed in Section 4.1, LibRISCV was originally intended for building custom ISA interpreters directly in Haskell. For this reason, it originally separated instruction decoding from instruction execution (i.e. the decoding is not part of the formal model; see Subsection 4.1.3.1). This can be illustrated by considering the formal description of the `LB` instruction in Listing 4.8 again. The semantics of this instruction are defined over a record type constructor (`LBInst`) in Line 1 which represents a decoded `LB` instruction. The different members of this record type are assigned to variables; the values of these variables correspond directly to integer values (e.g. 15 for accessing register `x15`) and are hence not captured by the formal description. To overcome this limitation, we added additional primitives to LibRISCV to express decoding operations as part of the instruction semantics descriptions. The resulting, enhanced description of the `LB` instruction is shown in Listing 4.9. Contrary to the description in Listing 4.8,

[19]Note that this is conceptually similar to the BinSym symbolic execution implementation presented in Subsection 4.2.1.2. In technical terms, both the symbolic execution as well as the code generation are just interpreters for the free monad construction provided by LibRISCV.

```
1 semantics LBOpcode = do
2   dest <- decodeRD
3   base <- decodeRS1 >>= readRegister
4   off  <- decodeImmI
5
6   byte <- loadByte (base `Add` off)
7   writeRegister dest (SExtByte byte)
```

Listing 4.9.: Description of the LB instruction with our LIBRISCV changes.

```
semantics LBOpcode = do
  (dest, base, off) <- decodeAndReadIType
  byte <- loadByte (base `Add` off)
  writeRegister dest (SExtByte byte)
```

Listing 4.10.: Final refinement of LB instruction semantics in LIBRISCV.

this version is only parameterized over the instruction opcode (`LBOpcode`) and then uses the new primitives `decodeRD`, `decodeRS1`, and `decodeImmI` to obtain additional information about the current instruction (Line 2 - Line 4). Since the description is now more verbose, we added an abstraction to define the instruction type, as mandated by the RISC-V specification [153, Section 2.2], as part of the formal description. The actual description of the `LB` instruction—using our enhanced version of LIBRISCV—is therefore less verbose and depicted in Listing 4.10. Notably, it has the same length as the original description (Listing 4.8).

The new instruction decoding primitives that we have added to the LIBRISCV ISA model allow us to map these to decoding functions provided by RISC-V simulators using our interface model. More details on interface modeling will be provided in the next subsection.

4.3.1.3. Interface Model

The interface model is the central prerequisite for generating a simulator-agnostic ISS as the generated implementation of instruction semantics will need to interface with existing components of a simulator (e.g. the register file). Since the C/C++ code—emitted by our code generation tool—should be simulator-agnostic, we introduce the interface model as an additional abstraction layer within the simulator. The interface model provides a generic C/C++ API for accessing the aforementioned components; this API is

```c
/* Register file */
uint32_t read_register(void *core, unsigned idx);
void write_register(void *core,
                    unsigned idx,
                    uint32_t value);

/* Byte-addressable memory */
uint8_t load_byte(void *core, uint32_t addr);
uint16_t load_half(void *core, uint32_t addr);
uint32_t load_word(void *core, uint32_t addr);
/* ... */
```

Listing 4.11.: Excerpt of the generic API provided by the interface model.

used by the code generator tool and needs to be implemented manually once for each targeted simulator. An excerpt of the generic API is shown in Listing 4.11, the full API description is available separately.[20] As illustrated in this figure, the API consists of a set of C functions which are parameterized over a void pointer. These void pointers are converted to simulator-specific types internally in the implementation of these functions. We decided against utilizing C++ abstractions (such as abstract classes) for this purpose to also support RISC-V simulators that are purely written in C. Presently, the generic API consists of 19 C functions and provides an interface for the register file, the program counter, the memory, and the decoder of a RISC-V simulator. Relying solely on a functional abstraction eases implementing this generic API as an implementation is essentially a mapping of the defined generic functions to simulator-specific ones. Therefore, these functions will be inlined by the C/C++ compiler in the common case and hence the additional interface model abstraction has minimal to no impact on simulation performance (see Subsection 4.3.2.3). We will further discuss the interface model implementation for Spike and `riscv-vp` in Subsection 4.3.2.1. In the following, we will introduce our simulator-agnostic code generation tool and illustrate how this tool interacts with the interface model.

4.3.1.4. Code Generation

We use the previously described ISA and interface models to implement a simulator-agnostic code generation tool. As depicted in Figure 4.5, the tool generates a simulator-

[20] `https://github.com/agra-uni-bremen/formal-iss/tree/fdl-2023#readme`

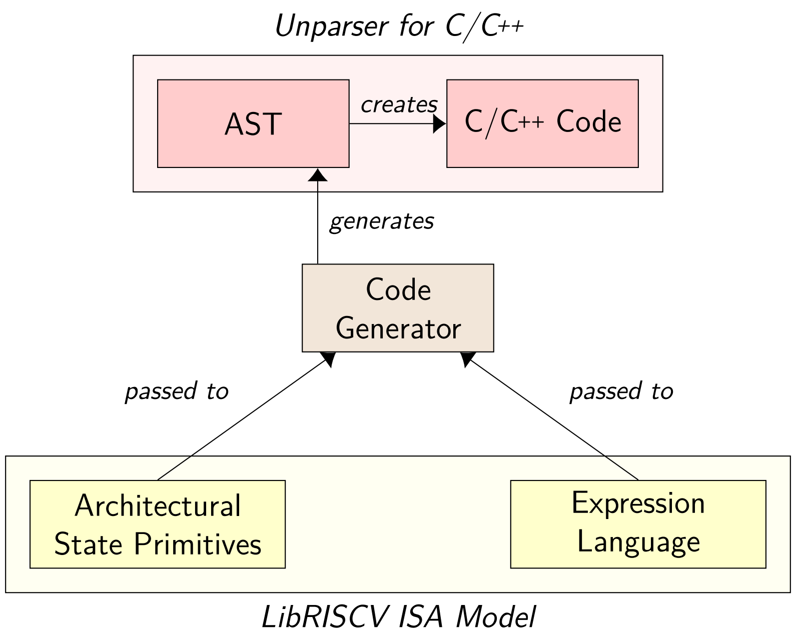


Figure 4.6.: Interaction between the code generator and the LibRISCV ISA model.

agnostic instruction execution unit, i.e. the code implementing the RISC-V instruction semantics. For this purpose, we build on the formal description of these semantics provided by LibRISCV. As discussed in Subsection 4.1.3, the formal ISA model consists conceptually of two components: primitives for describing interactions with the architectural state components and an expression language for describing operations on register/memory values that were obtained through these primitives. All instruction semantics are formally described using these components. In order to automatically generate code from this formal description, we need to build a code generator in Haskell which receives these EDSL components as inputs. As discussed in Subsection 4.1.3.3, the code generator then acts as an interpreter for the LibRISCV EDSL, transforming its components into the desired representation. As part of this transformation, we generate code for all 26 primitives of the EDSL, e.g. mapping the `readRegister` primitive to C/C++ code retrieving a register value through the interface model. Therefore, the desired representation is a C/C++ Abstract Syntax Tree (AST) in our case. The creation of this AST from the formal ISA model is illustrated in Figure 4.6.

As depicted in Figure 4.6, code implementing instruction semantics is created from this generated AST using an unparser (also called a pretty printer). Conceptually, an unparser is the opposite of a parser. As shown in Figure 4.6, it serializes a given AST to a chosen output format (C/C++ source code in our case) [94, 201, 87]. By employing an unparser, we can ensure the syntactic correctness of the generated code, compared to—for example—generating the code directly through string concatenation. This enables straightforward adjustments of the generated code and eases the application of our approach to simulators written in other programming languages. The implementation of the unparser (i.e. the translation from the AST to the C/C++ code) makes use of the existing `language-c`[21] Haskell library. As shown in Figure 4.6, our code generation tool is in this context responsible for generating an AST that is passed to the unparser provided by `language-c`. The generation of this AST is based on the formal instruction semantics obtained from LibRISCV.

As part of the AST generation, we create one C/C++ function for each formally described RISC-V instruction. As an example, the generated function that implements the `LB` instruction is shown in Listing 4.12. Each generated function receives a void pointer to a simulator-specific processor abstraction (`core`), the program counter of the current instruction (`instrPC`), and a void pointer to a simulator-specific instruction abstraction (`instr`) as function arguments. Naturally, since the code is automatically generated, it heavily nests function calls and is not optimized for human readability. Nonetheless, it is possible to illustrate the interaction with the aforementioned generic API of the interface model using this example. The function body shown in Line 5 - Line 8 of Listing 4.12 uses the `write_register`, `read_register`, and `load_byte` functions from the generic API (see Listing 4.11) to interact with the register file and memory implementation. These functions receive the processor abstraction (`core`) as a void pointer function argument and cast this pointer to a simulation-specific type internally to implement the operation.[22] Furthermore, the generated code in Listing 4.12 also obtains information about the instruction (register and immediate) using the `instr_rs1` and `instr_immI` functions of the interface model. Arithmetic operations are performed on these values by mapping the `Add` operation from LibRISCV's expression language (see Listing 4.10) to the `+` operator provided by C/C++. Similarly, the sign-extension from Listing 4.10 (`SExtByte`) is implemented in Listing 4.12 using integer type casts.

By leveraging the interface model, the code generation tool itself remains entirely simulator-agnostic. The tool is a standalone Haskell binary written in roughly 750 LOC which depends on the LibRISCV Haskell library (for the formal RISC-V model) and the `language-c` library (for C/C++ unparsing). In the next section, we illustrate that

[21]`https://hackage.haskell.org/package/language-c`

[22]Refer to Subsection 4.3.2.1 for more information on the simulator-specific implementation of the interface model for Spike and `riscv-vp`.

```c
static inline void exec_lb(void * core,
                           uint32_t instrPC,
                           void * instr)
{
  write_register(core, instr_rd(instr),
    (int32_t)(int8_t)load_byte(core,
      read_register(core,
        instr_rs1(instr))+instr_immI(instr)));
}
```

Listing 4.12.: Automatically generated C/C++ code for the LB instruction.

we can easily employ this tool—and our general approach—for automatically generating an execution unit for different existing RISC-V simulators, thereby demonstrating that minimally invasive ISS generation is possible.

4.3.2. Evaluation

In the following, we evaluate our approach in terms of generalizability, conformance, and simulation performance. In this regard, we have concerned ourselves with the following research questions: (a) is the approach generalizable in the sense that it can be applied to different RISC-V simulators? (b) does the generated ISS conform to the instruction semantics mandated by the RISC-V specification? (c) does the original, manually written, ISS have better simulation performance than the generated one? We present experiments, designed to answer these questions, in the following subsections. Each subsection concerns itself with one research question.

4.3.2.1. Generalizability

Our proposed ISS generation approach is specifically designed to be easily applicable to a variety of different RISC-V simulators. In order to evaluate the suitability of our approach for this purpose, we have employed it to generate a new ISS for the popular Spike [200] and `riscv-vp` [85] simulators. The existing ISS of these simulators was manually written by the developers in C++ and was not generated from a formal specification. In the following, we provide more background information on these two simulators and describe the changes that were necessary to integrate them with our ISS generation approach.

Spike was one of the first simulators for the RISC-V architecture and was initially developed by the University of California. Similar to `riscv-vp`, it simulates the execution of RISC-V machine code on a host system. In this regard, it focuses on achieving a high simulation speed at the cost of simulation accuracy. For this reason, it does not use a hardware modeling language like SystemC and therefore only has limited support for additional hardware peripherals. Contrary to Spike, `riscv-vp` provides a full virtual prototype of common RISC-V hardware platforms, including peripherals provided by these platforms. Additionally, `riscv-vp` focuses more on simulation accuracy and therefore also uses the SystemC hardware modeling language. The entire execution of RISC-V machine code is performed within a SystemC simulation, which eases reasoning about low-level details (e.g. timing). More information on `riscv-vp` is available in a publication by Herdt et al. [85]. We chose Spike and `riscv-vp` for our experiments because they represent two ends of a spectrum (Spike focuses on simulation performance while `riscv-vp` focuses on simulation accuracy) and their implementations therefore differ significantly. This allows us to demonstrate that our approach is applicable to a variety of existing simulators, from full VPs to performance-oriented simulators like Spike.

In order to employ our ISS generation approach for these simulators, we first had to manually implement an interface model for each simulator (see Subsection 4.3.1.3). As part of this implementation, we need to map the simulator-agnostic API for interacting with simulator components (e.g. the register file) to the internal simulator-specific API. An excerpt of the interface model implementation for `riscv-vp` is shown in Listing 4.13. As illustrated in this figure, the interface model casts a provided void pointer to a simulator-specific type for representing a RISC-V processor (`struct rv32::ISS`) and afterward calls methods of this type to implement the semantics of the interface model. The implementation presented in Listing 4.13 is specific to `riscv-vp` but the Spike interface model has a similar complexity. The complete implementation of both interface models is available as part of the artifacts. Apart from the interface model, we also had to connect the generated functions, which implement the semantics of RISC-V instructions (see Listing 4.12), with the existing fetch-decode-execute cycle implementations of Spike and `riscv-vp`. Spike already generates functions for the implementation of RISC-V instructions using build scripts, which we have adjusted accordingly. Contrary to Spike, `riscv-vp` uses a switch/case statement to execute decoded instructions, which—similar to Spike—we now generate using a script. In total, we modified roughly 150 lines in `riscv-vp` to implement the interface model and the build system changes. In Spike, we

```
static inline uint32_t
read_register(void *c, unsigned idx)
{
  return ((struct rv32::ISS*)c)->regs[idx];
}

static inline void
write_register(void *c, unsigned idx, uint32_t v)
{
  ((struct rv32::ISS*)c)->regs[idx] = v;
}

static inline uint8_t
load_byte(void *c, uint32_t addr)
{
  auto mem = ((struct rv32::ISS*)c)->mem;
  return mem->load_byte(addr);
}
```

Listing 4.13.: Excerpt of the interface model implementation for `riscv-vp`.

modified 200 lines for the same purpose.[23] The integration process took a programmer with domain knowledge less than a day. As such, the experiments demonstrate that minimal effort is required to apply our approach to different RISC-V simulators, thereby illustrating its generalizability.

4.3.2.2. Conformance

With the modifications outlined in the previous section, our enhanced versions of Spike and `riscv-vp` use an ISS that has been automatically generated from the LibRISCV ISA model, instead of a manually written one. Naturally, it is possible that the ISA model does not correctly capture the RISC-V instruction semantics or that our code generation tool or interface model implementations contain bugs. Therefore, it is paramount to ensure that the generated ISS still conforms to the RISC-V specification. In order to test conformance to the specification, we utilize the official RISC-V ISA tests for the

[23]Naturally, automatically generated lines are not included in this metric, since they do not correspond to any manual integration effort.

32-bit base instruction set.[24] These tests include multiple test programs (one per instruction) which validate the behavior of RISC-V instruction implementations using manually written test cases. Both our modified versions of Spike and `riscv-vp` pass the RISC-V ISA tests for `RV32I`. This indicates that our enhanced version of the LibRISCV ISA model still conforms to the RISC-V ISA tests and that our code generator and interface model do not introduce any severe bugs. In future work, we would like to expand our conformance tests by showing equivalence between the generated ISS and the manually written one.

4.3.2.3. Performance

Since our approach replaces a manually written ISS with an automatically generated one, there is the possibility that the code generation tool does not account for optimizations included in the manually written code. Simulation speed is of importance for RISC-V simulators in order to be able to execute and test complex RISC-V software in a reasonable time span. In order to evaluate the impact of our approach on simulation speed, we use our modified version of `riscv-vp` (see Subsection 4.3.2.1) and perform a simulation speed comparison with the original unmodified version of this simulator (referred to as the baseline version in the following). In prior work, performance benchmarks for `riscv-vp` have been conducted using the Embench benchmark suite [83]; therefore, we also use Embench for our experiments. Embench is an open source benchmark suite which is specifically tailored to the embedded domain, it consists of multiple benchmark programs which perform computation-intensive tasks (such as checksum calculation) [70]. We conduct our experiments with Embench 1.0 on a Linux system with an Intel i7-8565U processor.

Since benchmark results for a simulator can differ depending on the workload of the host system, we executed each benchmark application 25 times with both variants of `riscv-vp`. Benchmark results are presented as a grouped bar chart in Figure 4.7. The arithmetic mean of absolute execution time in seconds is given on the y-axis of Figure 4.7; the x-axis lists the benchmark programs of the Embench suite. For each benchmark program, two bar charts are presented: the left bar chart (blue) represents the results for the baseline version, the right bar chart (orange) represents our modified version of `riscv-vp` (i.e. uses an ISS generated using our approach). Both bar charts specify the arithmetic mean for the execution time of a given benchmark application

[24]https://github.com/riscv/riscv-tests

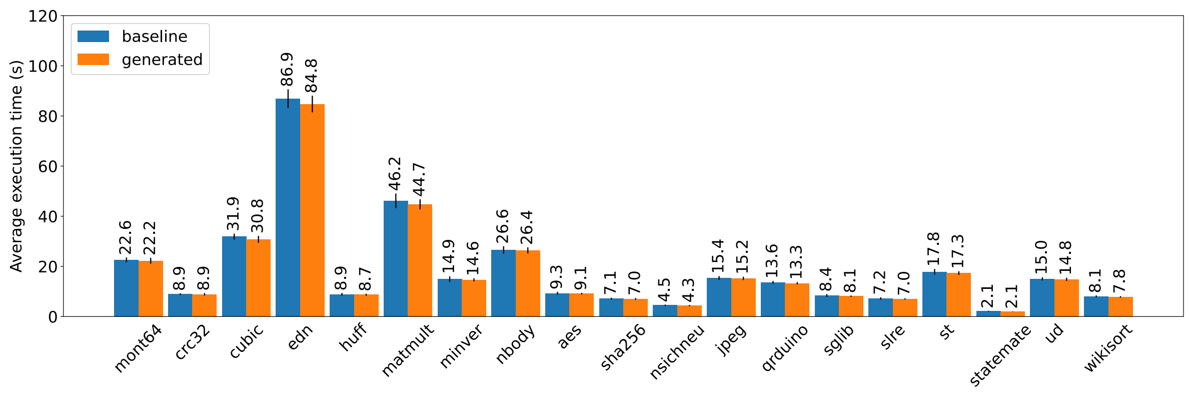


Figure 4.7.: Execution time benchmarks performed for an unmodified version of `riscv-vp` and a version generated using our approach. Each benchmark application has been executed 25 times. The bars represent the arithmetic mean over all executions (*lower is better*); error bars show the standard derivation.

over 25 executions. The error bars in Figure 4.7 specify the standard derivation.

In total, 19 benchmark applications have been tested with both variants of `riscv-vp`. Comparing the results for each benchmark application, the execution time for the generated `riscv-vp` variant is either slightly lower or the same as the execution time of the baseline version. This indicates that an ISS generated using our approach does not have worse simulation performance than a manually written one. Since the generated instruction semantics are the same for both Spike and `riscv-vp` (only the interface model differs), we do not provide a comparison of Spike variants in this thesis.

4.3.3. Related Work

We have already discussed related work regarding formal ISA models in Subsection 4.1.5. Additionally, we have examined prior work on the application of such models to binary analysis in Subsection 4.2.3. In this section, we focus on related work regarding code generation based on formal ISA models. In this regard, prior work by Reid has proposed deriving test suites and Verilog code for the ARM-v8 architecture from a custom formal model [150]. Similarly, prior work on Sail also supports code generation but operates on a higher abstraction level and supports a larger variety of use cases. Specifically, Sails supports generation of both executable code for different programming languages (C and OCaml) as well as generation of theorem proving languages (Coq, Isabelle, and HOL4) [5]. The key difference between our work and prior work (for which Sail is representative) is that the latter focuses on completeness and therefore goes beyond the description of instruction semantics. For example, Sail includes formalization of additional ISA details such as address translation algorithms or instruction decoding. This contributes to the complexity of these formal models and makes it difficult to integrate them with existing simulators. As such, Sail thus instead generates a new standalone ISA simulator [5, Section 5].

Apart from work on formal models, related work in the electronic design automation domain leverages Architecture Description Languages (ADLs) for processor descriptions [151, 67, 212]. Compared to formal ISA models, these ADLs focus more on microarchitectural details (such as pipelining or caching). For this reason, it is also challenging to integrate them with existing simulators and VPs. Therefore, similar to Sail, these languages are primarily used to generate new simulators instead of aiming for an integration with existing ones. To the best of our knowledge, the generation approach presented here is the first which is easily applicable to a variety of existing simulators, from full VPs like `riscv-vp` to performance-focused simulators like Spike.

4.3.4. Discussion and Future Work

In this section, we focused on a minimally invasive integration of formal ISA models with existing RISC-V simulators. Especially in the VP domain, formal models have not yet been used to their full potential. In order to ease usage of formal ISA models for VP generation, we focused on minimizing the integration effort. Our work towards this goal was motivated by enabling usage of our proposed symbolic execution approach (see Section 3.1 and Section 4.2) with additional VPs and simulators. However, so far, we have only generated an ISS which performs concrete execution. Nonetheless, the approach can be easily extended to also generate a symbolic ISS; we believe this to be primarily an engineering effort. Due to this underlying motivation, we focused primarily on generating code implementing the instruction semantics. In future work, it would be possible to extend the formal model to cover more parts of the ISA (e.g. memory behavior or decoding). This would make our code generation approach more comprehensive. However, in this regard, there is a trade-off between the comprehensiveness of the formal model and the effort required to integrate it with existing simulators. The premise of our work is that by focusing on the code implementing the actual instruction semantics—where the majority of changes for an integration with symbolic execution occur—we can ease the integration with existing simulators.

Additionally, in terms of comprehensiveness, we only focused on the 32-bit RISC-V base instruction set in this section. As discussed in Subsection 4.1.6, it would be possible to extend the underlying LibRISCV ISA model to support additional RISC-V extensions and RISC-V variants (e.g. 64-bit) in future work. In Subsection 4.2.2.1, we have demonstrated that it is possible to model additional extensions (specifically the M-extension) using LibRISCV, but we have not yet integrated this extension with our code generation approach. In terms of additional extensions, it would be especially interesting to also support parts of the RISC-V privileged architecture specification [154], instead of concentrating on the user-level ISA [153].

4.3.5. Conclusion

In this section, we have presented a novel approach for generating the ISS of RISC-V simulators from a formal ISA model. Contrary to prior work, our approach is designed to be as minimally invasive as possible through a simulator-agnostic interface model (Subsection 4.3.1.3), a self-contained formal ISA model (Subsection 4.3.1.2), and a code generator for this model (Subsection 4.3.1.4). By focusing exclusively on the code implementing the actual instruction semantics, we can prospectively ease the integration of our symbolic execution approach with additional simulators and VPs. Conducted experiments confirm that our code generation approach is applicable to different RISC-V simulators (Spike and `riscv-vp`) with minimal effort (Subsection 4.3.2.1). Furthermore, we were able to show that an ISS—generated using our approach—still passes the RISC-V ISA tests (Subsection 4.3.2.2) and offers similar simulation speed performance as a manually written one (Subsection 4.3.2.3). In future work, we want to generate both a concrete and a symbolic ISS and, more generally, consider an application of our approach to additional VP-based firmware analysis tasks. To stimulate further research in this direction, we have released our code generation tool[25] as well as our modified versions of Spike[26] and `riscv-vp`[27] as open source software.

[25]`https://github.com/agra-uni-bremen/formal-iss`
[26]`https://github.com/agra-uni-bremen/spike-libriscv`
[27]`https://github.com/agra-uni-bremen/libriscv-vp`

Chapter 5.

Error Detection Techniques for Firmware Testing

In Chapter 3 and Chapter 4 we have presented a symbolic execution approach which is capable of analyzing low-level firmware for embedded devices. Based on a specific hardware peripheral input source (e.g. a UART), we can now enumerate execution paths that are reachable in the firmware through this input source by injecting a symbolic value using our SystemC TLM extension. Additionally, we need to check the properties of each executed path in order to uncover bugs in the tested software. Prior work uses the term *path analyzer* to refer to the component that is responsible for checking each path [41]. While this prior work has also presented multiple path analyzers, these existing analyzers are tailored to the conventional domain and designed to be used with operating systems such as Microsoft Windows. In this chapter, we therefore present novel path analyzers which are specifically designed for the detection of programming errors in firmware for embedded devices.

As discussed in Section 2.1, embedded devices are severely constrained regarding available computing resources (e.g. memory) [24]. Due to these constraints, embedded devices are often programmed using the C programming language [79, Table 1]. The C programming language is a popular choice for programming these devices as it gives programmers control over low-level machine details, thereby enabling optimization to reduce the use of scarce resources. Unfortunately, C is an inherently unsafe programming language, i.e. the C language specification leaves some behavior undefined. In this case, an attacker can rely on specifics of the utilized compiler and hardware platform to exploit the undefined behavior. The textbook example in this regard are memory corruptions caused by the lack of memory safety in the C programming language (e.g. buffer overflows or use-after-frees) [28]. Prior work by Szekeres et al. provides a detailed systematization of knowledge regarding the exploitation of such memory corruptions [181].

Detecting memory corruptions during automated firmware testing using symbolic execution is vital, as they can be trivially exploited by an attacker. Unfortunately, it is challenging to detect memory corruptions in embedded firmware. Conventional operating systems (e.g. Linux, BSD, or macOS) provide a variety of protection mechanisms which cause software to crash on many memory corruptions. For example, when the memory corruption leads to a violation of memory protections as enforced by a Memory Management Unit (MMU). As discussed in Chapter 1, embedded systems are often lacking these techniques because hardware features (like an MMU) are not widely available. Therefore, the majority of memory corruptions occur silently and do not result in an observable crash [128]. Additionally, development tools which utilize software instrumentation to ease detection of memory corruptions and similar programming errors (e.g. Valgrind [133] or AddressSanitizer [163]) are tightly integrated with conventional operating systems and—at the time of writing—do not support embedded systems [128, p. 4].

In order to overcome this challenge, we contribute three path analyzers to detect memory corruptions in embedded firmware during automated software testing with symbolic execution. The first analyzer presented in Section 5.1 relies on a tight integration with the virtual hardware (the VP) and detects memory corruptions during the execution of instructions which interact with the memory (e.g. `LW` or `SW`). For this purpose, the analyzer requires instrumentation of the tested firmware through a compiler pass. Such instrumentation may not always be possible, for example when the source code is not available. Therefore, Section 5.2 presents an analyzer which does not require instrumentation but can only detect a particular kind of memory corruption: stack overflows. On embedded systems, call stacks are typically statically allocated and do not grow dynamically. Hence, it is possible to exhaust the stack, thereby causing a memory corruption [149]. Detecting stack overflows requires us to reason about memory consumption; thus, this path analyzer also allows us to estimate stack memory usage of executed paths. Memory usage is an important characteristic in the embedded domain as utilized devices are severely constrained in terms of available memory (see Section 2.1). However, as symbolic execution is commonly not complete, both analyzers are intended for testing purposes since we cannot prove the absence of memory corruptions. In order to fully prevent such corruptions from occurring in a production environment, prior work has proposed the use of safer programming languages which offer memory safety (i.e. guarantee the absence of memory corruptions) through runtime bounds checks [114, 48, 137]. Prior work on safe C dialects attempts to ease the application of such language-based techniques to existing C code by retrofitting these safety features on top of the C programming language. Nonetheless, employing these dialects requires conversions and annotations of the existing C source code [65, 47, 99]. In Section 5.3 we provide evidence that symbolic execution can aid in this conversion process and that these dialects make previously silent memory corruptions observable. We evaluated our path analyzers by conducting experiments with the RIOT [9] operating system in which we uncovered 13 previously unknown bugs which have been reported to and acknowledged by RIOT developers. We choose RIOT for our experiments as a survey by Hahm et al. considers it to be the "most prominent open source OS" with multithreading support in the constrained embedded domain [79, Section 7.3.2]. The ability to identify memory corruptions in this popular operating system illustrates the effectiveness of our proposed path analyzers.

5.1. Detection of Spatial Memory Safety Violations

This section concerns itself with the general detection of *spatial violations* (e.g. buffer overflows) which can lead to memory corruptions, and—in the worst case—allow an attack to gain remote code execution. A spatial violation occurs when a pointer dereference is not within the memory allocated originally allocated to that pointer. The term *spatial memory safety* refers to a guaranteed absence of such spatial violations [65, Section 1]. The C/C++ family of programming languages, which is popular in the embedded domain, does not provide spatial memory safety. Due to the popularity of this programming language family, a large body of prior work concerns itself with the detection of spatial violations, some of these prior works even attempts to achieve spatial memory safety [133, 163, 131, 65, 60]. Unfortunately, all of these approach come with drawbacks (changes to the in-memory representation of pointers, additional runtime overhead, or manual effort).

In recent years, techniques which achieve spatial memory safety in hardware—instead of software—have yielded promising results and are gaining traction [60, 130, 206]. A detailed overview of prior work in this regard is provided by Jero et al. [98]. In this domain, early work on HardBound has achieved spatial safety by tracking bounds information for pointers in hardware and propagating them for instructions that are used to implement pointer arithmetic [60]. Every time a pointer is dereferenced through a load/store instruction, the bounds information can be consulted to check if the dereference is still within the bounds of the memory object originally allocated to that pointer. Inspired by prior work on Mondrian [205], more recent work on CHERI [206] generalizes this concept by allowing memory protection at a byte-granularity. In this context, achieving spatial memory safety is just an application of this concept [206, Section 5.1]. CHERI has been adopted by ARM as part of their Morello research prototype and is therefore attracting a growing community of researchers [203]. Unfortunately, similar to the techniques referenced in the prior paragraph, hardware-based approaches come with a drawback: they need custom hardware, and outside of Morello (which is still a prototype) this hardware is presently not available.

However, in the embedded domain employment of custom hardware with custom domain-specific instructions is common [51, Section 2]. Furthermore, approaches which detect spatial violations in hardware integrate well with our VP-based symbolic execution approach (Section 3.1) as we can easily implement these approaches in the virtual hardware provided by the VP. Doing so allows us to check each path executed by our VP-based symbolic execution engine for spatial violations. Conceptually, we are therefore facilitating a hardware-based approach for achieving spatial memory safety as a path analyzer for symbolic execution. For this purpose, we contribute a HardBound implementation modeled in SystemC TLM in this section. We chose HardBound over CHERI as we deemed it easier to implement, and we are specifically interested in spatial safety and not in the more general concept of byte-granular memory protections. HardBound requires a custom compiler pass to communicate initial bounds information from the software to the hardware. Unfortunately, the compiler pass implementation from the original HardBound paper is not open source; thus, we also contribute the first open source implementation of a HardBound compiler pass. In order to evaluate our implementation, we perform experiments with RIOT [9] where we uncover seven previously unknown spatial violations. To stimulate further research on this topic, we have released both our compiler pass as well as our VP-based HardBound implementation on GitHub.[1,2]

5.1.1. Background

The following subsections serve as a brief primer on the general theoretical concept of memory safety and HardBound as a technique for achieving memory safety in hardware.

[1] https://github.com/agra-uni-bremen/hardbound-llvm
[2] https://github.com/agra-uni-bremen/hardbound-vp

5.1.1.1. Memory Safety

Existing publications on memory safety issues of the C/C++ programming language family distinguish spatial memory safety and temporal memory safety as follows [65, Section 1]:

> *Temporal safety* is ensured when memory is never used after it is freed. *Spatial safety* is ensured when any pointer dereference is always within the memory allocated to that pointer.

The C/C++ programming language family offers neither spatial nor temporal memory safety. Lack of memory safety allows attackers to perform unintended computations which are central to many security vulnerabilities [28]. Prior work by Szekeres et al. provides a formal model for attacks exploiting memory safety issues [181]. A variety of different techniques have been proposed by prior work which attempt to address the lack of memory safety in the C/C++ programming languages [65, 60, 131]. As explained in the Section 5.1, our proposed approach is based on a combination of HardBound and symbolic execution, the former will be further described in the following.

5.1.1.2. HardBound

HardBound enforces spatial memory safety for C programs through a hardware peripheral. Enforcement is achieved by enhancing values representing C pointers with bounds information tracked in hardware. Contrary to software-only approaches, HardBound does not modify the C pointer representation and allows bounds checks to be performed efficiently in hardware. Conceptually, each register and memory value in HardBound is a triplet $(value, base, bound)$ where $value$ represents the original pointer value and $base/bound$ represent the lower/upper bound of the bounded pointer. We will refer to the additional *base* and *bound* information as HardBound metadata in the following. The HardBound metadata is used by a custom hardware peripheral to perform boundary checks on each load/store instruction. As such, it is ensured that each load/store is within the pointer bounds as specified by the *base* and *bound* metadata. For this reason, spatial violations cannot occur on bounded pointers [60].

Since the C type system is inaccessible at the binary level, the executed software must communicate which values represent pointers (and their respective HardBound metadata) to the hardware peripheral. In the original HardBound paper, this is achieved through a custom `setbound` instruction. The insertion of `setbound` instructions is auto-

```
1 load_addr R2, 0x1000
2 setbound R2, 0x1000, 4
3
4 load_byte R3, R2 # load at 0x1000, success
5 addi R2, R2, 2   # R2: (0x1002, 0x1000, 0x1004)
6 load_byte R3, R2 # load at 0x1002, success
7 addi R2, R2, 2   # R2: (0x1004, 0x1000, 0x1004)
8 load_byte R3, R2 # load at 0x1004, fail
```

Listing 5.1.: HardBound example usage [60, Figure 2].

mated using "simple intra-procedural compiler instrumentation" [60, p. 103]. The hardware peripheral is in turn responsible for propagating metadata initial set by the executed software. As an example, consider pointer arithmetic as performed using an `addi` instruction in the pseudo assembler code in Listing 5.1. In this example, a pointer to a four byte value at address `0x1000` is created (Line 1 - Line 2). The pointer is then incremented (Line 5 - Line 7) and dereferenced (Line 4 - Line 6), on each increment the metadata must be propagated. Ultimately, the load in Line 8 fails as the pointer value is no longer within the propagated bounds.

5.1.2. VP-based HardBound Integration

In the following subsections, we will describe how HardBound support can be integrated into a standard SystemC TLM architecture. For this purpose, we build upon the SymEx-VP symbolic execution engine described in Chapter 3. This VP targets the RISC-V architecture and allows symbolic execution of 32-bit RISC-V machine code.

5.1.2.1. Overview

Figure 5.1 provides an overview of our proposed architecture. As explained in Section 3.1, both the ISS and the symbolic execution engine are already provided by SymEx-VP. The latter provides us with symbolic types, a path explorer for finding new paths through the executed software, and an SMT solver for solving constraints on symbolic types. In order to employ HardBound as a path analyzer, we had to integrate it with the symbolic ISS provided by SymEx-VP. By relying on implicit C++ type conversion, we were able to keep required SymEx-VP modifications to a minimum, ultimately only extending around 700 LOC. This illustrates that a non-intrusive integration is possible.

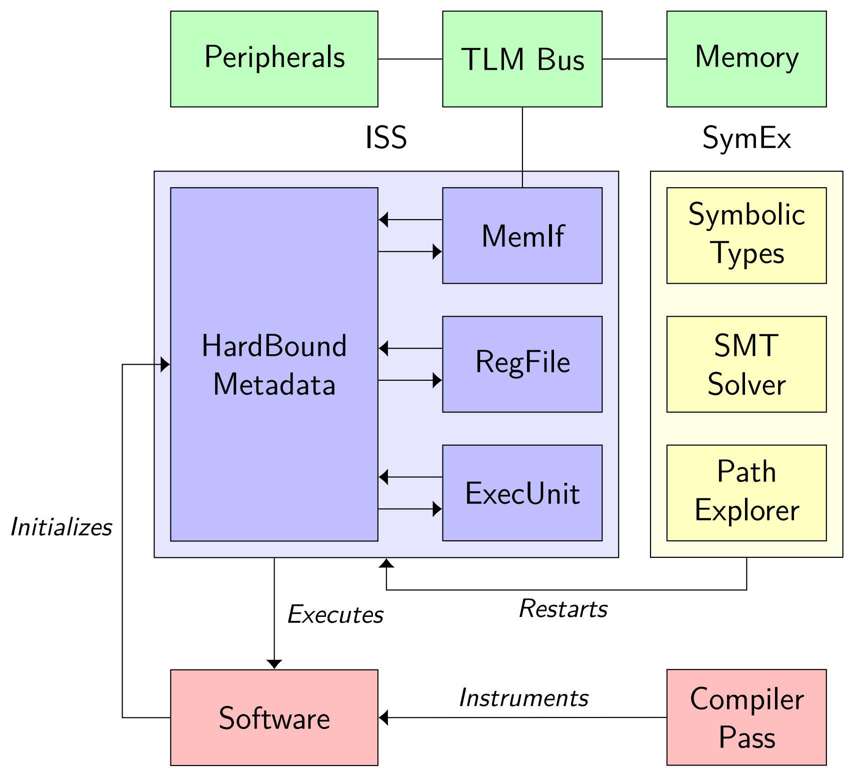


Figure 5.1.: Overview of our HardBound implementation for SymEx-VP.

In the context of the HardBound integration, the ISS is the main component of the VP as it is responsible for fetching, decoding, and executing RISC-V instructions. As such, it executes a given RISC-V software provided in binary form. Execution is based on symbolic input values, supplied by the symbolic execution engine (referred to as SymEx in Figure 5.1). After simulation terminates, the symbolic execution engine determines new assignments for symbolic input variables and restarts the ISS—and thereby also the software—with these new variable assignments. New variable assignments result in the discovery of new paths through the executed software. On each path, performed load/store instructions are bounds checked using the provided HardBound metadata (see Subsection 5.1.1.2). This metadata is initialized by the executed software through special instructions inserted by a compiler pass. The ISS is responsible for propagating it correctly during execution, as previously illustrated in Subsection 5.1.1.2.

5.1.2.2. Metadata Propagation

Since our work is based on SymEx-VP, our ISS supports symbolic execution. As such, instruction operands may represent symbolic values. We modified the ISS to also track and propagate HardBound metadata alongside these symbolic values. During software execution, the ISS interacts with the stored metadata to perform bounds checks. Recall that HardBound metadata is only required for values representing C pointers. These values may be stored in either memory or registers. As such, the register file (RegFile) and the memory interface (MemIf) had to be modified (see center of Figure 5.1). The register file is responsible for storing register values. The memory interface is responsible for interactions with memory-mapped peripherals through the SystemC TLM bus. We modified both—the memory interface and the register file—to ensure they store associated HardBound metadata for memory and register values representing pointers. The main execution unit (ExecUnit) of the ISS accesses this metadata and propagates it when executing instructions which are used to implement C pointer arithmetic. For example, an instruction adding a constant to a bounded pointer value must itself return a bounded pointer value with associated HardBound metadata (see Listing 5.1).

Implementation of metadata propagation was the most invasive change made to SymEx-VP as we had to switch the underlying data type, used by the execution unit for instruction operands, from symbolic expressions to a tuple which additionally tracks the HardBound metadata. The modifications required for this change were kept to a minimum by relying on implicit C++ type conversions, thereby allowing implicit conversions from symbolic expressions—with associated HardBound metadata—to plain symbolic expressions. As the majority of RISC-V instructions are not commonly used to manipulate pointer values, we were able to refrain from modifying the implementation of these instructions. As such, only the implementation of instructions which are used by C compilers to implement pointer arithmetic had to be modified in the execution unit. For the RISC-V architecture, we thus implemented metadata propagation for the following instructions: `ADD`, `ADDI`, and `SUB`. In the following subsection, we will explain how propagated metadata interacts with SystemC TLM in the HardBound context.

5.1.2.3. TLM Integration

Storing and propagating HardBound metadata in the ISS allows us to perform bounds checks on load/store instructions. In the SystemC context, these instructions are implemented through SystemC TLM based on a bus abstraction. The ISS communicates with devices attached to the TLM bus (e.g. memory or memory-mapped peripherals) using TLM transactions created by a memory interface. In order to avoid modifications of peripherals attached to the TLM bus, we are not propagating HardBound metadata over TLM and instead perform a transparent conversion within the memory interface itself using an internal mapping $\delta\colon addr \mapsto \{(base, bound)\}$. HardBound metadata ($base$ and $bound$) can be updated by storing new metadata at $addr$ using a store instruction. Load instructions return HardBound metadata if a mapping $addr \rightarrow (base, bound)$ exists in δ for the loaded $addr$. HardBound metadata originates in the executed software through custom `setbound` instructions. While the ISS is responsible for the storage and propagation of HardBound metadata, it is incapable of initializing the metadata, as information regarding pointer bounds is difficult to infer at the binary level. Therefore, the software itself communicates initial metadata values to the ISS upon pointer creation. This is achieved through the aforementioned `setbound` instructions. For each created pointer, these instructions are automatically inserted into the tested software by a compiler pass.

5.1.2.4. Compiler Pass

Conceptually, the compiler pass performs an analysis detecting the creation of pointers, infers the size of the values pointed to, and communicates this information to the VP. In order to ease supporting MMIO, we deviated from the compiler pass implementation in the original HardBound paper regarding the handling of pointer casts. Normally, creating a pointer from an integer (e.g. `(int *)0x1000`) is an unsafe operation as no bounds information is associated with the memory address `0x1000` [60, Section 6.1]. This is, however, a common idiom to communicate with memory-mapped peripherals from low-level C code. For this reason, we relaxed the handling of these unsafe casts in the compiler pass. This prevented the addition of manual `setbound` invocations for memory-mapped peripheral accesses.

Furthermore, the original HardBound paper does not provide a detailed description of how direct array accesses are handled in the compiler pass. Consider an array access such as `buf[i] = n`, since no pointer exists in this example code, no HardBound metadata is available for ensuring spatial memory safety. To mitigate this problem, we have written two compiler passes. The first transforms any array access of the form `buf[i] = n` to a pointer-based access of the form `*(buf + i) = n`. The second pass communicates pointer bounds, upon pointer creation, to the VP.

Listing 5.2 illustrates the transformations performed by the two compiler passes. The original code (Line 9) performs a direct array access on `buf`. The first compiler pass rewrites this to a pointer-based access (Line 11 - Line 12). The second compiler pass inserts an appropriate call to a `setbound` function which communicates the bounds of `ptr` to the VP (Line 14 - Line 16).[3]

[3]On a technical note, we intercept RISC-V `ecall` instructions in the VP to set bounds information

```c
static char buf[BUFFER_SIZE];

int add_to_buffer(char c) {
  static size_t index = 0;
  if (index >= BUFFER_SIZE)
    return -1;

  // ------- [[ Original Code ]] -------
  buf[index] = c;
  // ------- [[ 1st Compiler Pass ]] -------
  char *ptr = &buf[0];
  *(ptr + index) = c;
  // ------- [[ 2nd Compiler Pass ]] -------
  char *ptr = &buf[0];
  setbound(&ptr, ptr, sizeof(buf));
  *(ptr + index) = c;
  // ------- END -------

  index++;
  return 0;
}
```

Listing 5.2.: Performed HardBound compiler pass transformations.

5.1.3. Evaluation

We evaluate our approach by applying it to the RIOT operating system. We used RIOT as an evaluation target since prior work by Hahm et al. considers it to be one of the "most prominent open source" operating systems with multithreading support in this domain [79, Section 7]. Furthermore, RIOT employs a code quality management process and automated unit tests, thereby aiming for high code quality [9, Section 14]. This allows us to evaluate whether our approach is capable of finding real bugs missed during manual code review and unit testing. RIOT supports a variety of hardware platforms. For our experiments we utilize the constrained SiFive HiFive1 platform which uses RISC-V and is supported by both RIOT and SymEx-VP. RIOT itself is further described in a publication by Baccelli et al. [9].

from the software. The original HardBound paper uses custom instructions.

In accordance with prior work, we believe input handling routines of the network stack to be the biggest attack vector of a networked IoT operating system [159]. Our experiments therefore focus on RIOT components which are part of this network stack. In the following, we will further describe how we employed HardBound in the RIOT context to analyze these components and which memory safety violations we were able to uncover through our analysis.

5.1.3.1. RIOT HardBound Setup

HardBound is intended to be deployable with “minimally invasive changes to the compiler and runtime” [60, Section 3.2]. In the following, we describe the necessary changes for an integration with RIOT. As a first step, we had to ensure that the RIOT build system utilizes our LLVM-based compiler pass. Fortunately, RIOT already supports compilation with LLVM. For this reason, we only had to modify the employed compiler flags via a build system configuration variable. The original HardBound paper also acknowledges that library functions performing memory allocations, e.g. `malloc`, need to be modified to include appropriate `setbound` calls [60, Section 3.2]. While usage of `malloc` in RIOT is discouraged, we still had to modify the RIOT module used to allocate memory for network packets to set the appropriate bounds for each returned packet. This allows us to discover spatial violations potentially occurring when accessing these packets.

Overall, our HardBound setup for the RIOT operating system was straightforward and only required the outlined changes, which we believe to be negligible in terms of effort required. For our experiments, we boot RIOT on our HardBound extended VP and perform boundary checks in the VP during the execution of RIOT software. More details are provided in the next subsection.

5.1.3.2. Results

RIOT follows a modular software architecture, modules which should be enabled are selected at compile-time [9, Section 4]. We tested multiple RIOT modules which are part of the network stack using our proposed software testing technique. Similar to the prior evaluation conducted in Subsection 3.1.5, we distinguish two test types:

1. UNIT tests, conducted using custom test drivers which invoke functions from the public module API. In this case, symbolic values are created directly in the test driver.

Table 5.1.: Spatial memory safety violations found in RIOT modules.

Id	Module	Test	#Paths	Time	#instr
#15927	`uri_parser`	UNIT	48	11 s	408646
#15930	`uri_parser`	UNIT	156	35 s	1311034
#15945	`clif`	UNIT	227	50 s	1847765
#15947	`clif`	UNIT	10	2 s	91302
#16018	`gnrc_rpl`	SLIP	75	143 s	3378636
#16062	`gnrc_rpl`	SLIP	72	307 s	3444769
#16085	`gnrc_rpl`	SLIP	855	3532 s	46262378

2. SLIP tests, conducted using existing RIOT example applications. Symbolic values are introduced through a custom SLIP [155] network interface, which is implemented in the VP.

We tested different modules of the RIOT network stack using both approaches. We used UNIT tests for utility modules, used indirectly by network protocol implementations. SLIP tests were used for freestanding implementations of network protocols, which directly process input received through the IP. In total, we tested the following three RIOT network modules which implement internet protocols that we believe to be in common use in the low-end IoT context on constrained devices:

1. The `uri_parser` module, which provides a non-destructive parser for Uniform Resource Identifier (URI) references as defined in RFC 3986 [17].

2. The `clif` module, which provides a parser for the CoRE Link Format as used in REST architectures for constrained devices and defined in RFC 6690 [166].

3. The `gnrc_rpl` module, which provides an implementation of the Routing Protocol for Low-Power and Lossy Networks (RPL) as defined in RFC 6550 [2].

In these modules, we found seven previously unknown spatial memory safety violations, all of which have been discovered using our proposed combination of symbolic execution

Table 5.2.: Description of spatial safety violations found in RIOT network modules.

Module	Bug Description
`uri_parser`	**#15927:** During parsing of the *userinfo* part of a URI the parser did not check if the input is long enough to even contain a complete *userinfo* if the URI contained an `@` character, thereby performing an out-of-bounds read on the provided input buffer. **#15930:** The `uri_parser` module attempted to parse data after the *hier-part* of the URI, even if none was present. For example, on an input like `a://` the parser would perform an out-of-bounds read due to missing bounds checks.
`clif`	**#15945:** During parsing of key-value pairs, the module did not check if a value was actually present after the key has been read, thereby performing an out-of-bounds read. **#15947:** During parsing of link attributes, the `clif` module did not check whether any attributes were present, thus performing an out-of-bounds read on the input buffer.
`gnrc_rpl`	**#16018:** RPL messages are parsed by casting buffers to packed structs. Unfortunately, `gnrc_rpl` did not check if the buffer was large enough to contain the struct in some instances. **#16062:** During validation of RPL options, `gnrc_rpl` did not check if the input is large enough to contain a given option. Attempts to access this option resulted in an out-of-bounds read. **#16085:** The `gnrc_rpl` module separates option parsing from option validation. Similar to #16062, the option parsing code was also lacking proper bounds checks, resulting in an out-of-bounds read.

and HardBound. These findings were reported to and acknowledged by RIOT developers. Further information regarding the discovery of individual issues is provided in Table 5.1. For each discovered issue, we list the identifier in the public RIOT issue tracker[4], the module in which it was found, the employed test method, the number of paths enumerated until it was found, and the total execution time. As a complexity metric, we also include the total amount of RISC-V instruction executed.

In modules which we tested through UNIT tests, spatial memory safety violations are discovered faster, and fewer instructions are executed. This is due to the fact, that fewer constraints are tracked as we only test individual functions. With SLIP tests, issue discovery takes longer as the input is passed through the entire network stack, thereby imposing more constraints on symbolic input variables. However, as discussed

[4]`https://github.com/RIOT-OS/RIOT/issues`

in Subsection 3.1.5, SLIP-based tests allow us to test an entire application as-is, thereby reducing the integration effort. Additionally, given the complexity of routing protocols (such as RPL), this indicates that our approach is also capable of finding spatial memory safety violations in complex real-world code for constrained devices.

5.1.4. Related Work

Prior work has primarily focused on the detection of spatial memory safety violations on conventional devices. Popular approaches in this regard include EXE [34], which utilizes compiler instrumentations to detect spatial memory safety violations, and KLEE [32] which symbolically executes LLVM IR, the intermediate language used by the LLVM compiler infrastructure. As such, KLEE-based approaches do not capture low-level machine details and operate on a higher abstraction level than our approach, which symbolically executes machine code directly. As discussed at length in Section 4.2, this has multiple drawbacks, including decreased symbolic execution performance (see Subsection 4.2.2.2).

With the focus on conventional systems, the aforementioned publications also do not address challenges specific to constrained embedded devices. Prior work by Muench et al. discusses these challenges further [128]. They identified “silent memory corruptions” as the predominant issue in this domain, most of which would be detected by conventional operating systems through employed protection mechanisms lacking on embedded devices to reduce production costs [128, Section 3]. They also propose heuristics to improve error detection in this domain and combine these heuristics with fuzzing to automatically discover memory corruptions [128, Section 6]. Our approach does not rely on heuristics and we believe symbolic execution to be preferable over fuzzing on embedded devices as the state space is smaller due to limitations on code size. This might mitigate the state explosion problem known from conventional devices [10, Section 1.2].

This hypothesis is confirmed by Davidson et al. who present FIE, a symbolic execution framework for finding vulnerabilities in embedded firmware [54]. FIE targets MSP430 microcontrollers and attempts to achieve a “complete analyses for simple firmware programs” [54, Section 3]. Specifically, this prior work was able to fully verify memory safety for 53 of 99 firmware images in their test corpus [54, Section 1]. FIE is based on KLEE and therefore also executes LLVM IR symbolically. For this reason, FIE operates on a higher abstraction level than our own approach, which symbolically executes RISC-V machine code directly. This comes with the drawback that FIE cannot take low-level machine details into consideration. As an example, FIE fails to execute paths which include inline assembly, usage of which we believe to be common on embedded devices (see Section 4.2). Furthermore, FIE does not use accurate models of hardware peripherals. Instead, it approximates peripheral behavior through given memory and interrupt specifications, which results in potential false-positives [54, Section 6]. Since our approach is based on SystemC, we have existing models of peripherals at our disposal and can easily model new ones (see Chapter 3).

Lastly, prior work by Herdt et al. provides symbolic execution of embedded binaries [84]. However, this publication does not support unmodified SystemC peripherals. Furthermore, the employed path analyzer is only capable of detecting spatial violations in buffers allocated dynamically through `malloc`. Contrary to our own approach, this publication therefore misses overflows in data structure not allocated dynamically (e.g. buffers allocated on the stack) [84, Section 4.4.2].

5.1.5. Conclusion

In this section, we concerned ourselves with the early detection of spatial memory safety violations on constrained embedded devices. We proposed and implemented a VP-based software testing technique for this purpose which uses symbolic execution to enumerate reachable program paths and checks each path for spatial violations through a SystemC-based HardBound implementation (Subsection 5.1.2). We illustrated an architecture for achieving a non-intrusive integration of HardBound with existing SystemC-based VPs and implemented this architecture on top of SymEx-VP. We have also released our HardBound extended version of SymEx-VP on GitHub to stimulate further research. Our main contribution is a new path analyzer for an existing symbolic execution engine. As per Subsection 5.1.4, our approach is novel as it addresses challenges specific to embedded devices not addressed in prior work. We applied our approach to an operating system for constrained embedded devices (RIOT), where we found seven previously unknown bugs, which had been missed by unit testing and manual code review (Subsection 5.1.3).

5.2. Minimally Invasive Stack Overflow Detection

In Section 5.1, we presented a path analyzer for the detection of spatial memory violations to prevent memory corruptions from occurring in a production environment. The path analyzer required excessive instrumentation and minor modifications of the tested firmware. Such modifications may not always be possible, e.g. if the source code is not available. Furthermore, the instrumentations are specifically intended for testing purposes only as hardware with HardBound support is presently not available. Meaning, the tested firmware differs from the one deployed in a production environment, thus potentially causing divergent behavior to be observed during testing. In this section, we present a path analyzer which does not require any modifications of the tested firmware. However, as a trade-off, we only focus on the detection of a particular kind of memory corruption in this section: stack overflows (i.e. overflows of the call stack).

As per Section 2.1, memory is severely limited on embedded devices. In order to avoid fragmentation of available memory, firmware for constrained devices does not use dynamic memory allocation but instead allocates all memory statically. In a multithreaded environment, this creates an interesting problem regarding the allocation of stack space for different threads executed by the firmware. As the stack space is also statically allocated at compile-time, the programmer must choose an appropriate maximum stack size before deploying the software. If the code executed in a given thread does not use the allocated stack space in its entirety, memory is wasted thus potentially causing increased production costs. If the stack size—chosen by the programmer—is too small for the thread using it, a stack overflow might occur. Stack overflows are a kind of memory corruption [181], similar to other memory corruptions, they occur silently on embedded devices [128]. In the worst case, an undetected stack overflow might allow an attacker to subvert program control flow and achieve remote code execution.

In order to detect stack overflows during automated testing with symbolic execution, we need a path analyzer which reasons about stack memory usage of each executed path. For this purpose, we built upon prior work by Park et al. which proposes a stack size estimation and stack overflow detection technique [136]. The implementation presented in the aforementioned prior work requires custom compilers and software instrumentation. We improve upon this technique by illustrating that sanity checks originally performed in the software—using code inserted by a custom compiler—can also be performed in the execution environment. Thereby allowing the analysis to be performed without any instrumentation or modification of the executed software. In contrast to prior work on stack overflow detection using formal methods [149, 101, 36], we focus on bug hunting using symbolic execution and thereby ease application to existing software. We demonstrate the effectiveness of our approach by performing experiments with RIOT where we identified two previously unknown stack overflows under specific configurations.

```
1:  procedure CHECK_STACK($func$)
2:      $thread \leftarrow current\_thread()$
3:      $required \leftarrow current\_stackuse + func.stackuse$
4:
5:      if $required > thread.stacksize$ then
6:          $handle\_stack\_overflow()$
7:      else if $required > max\_stackuse[thread]$ then
8:          $max\_stackuse[thread] \leftarrow required$
9:      end if
10: end procedure
```

Figure 5.2.: Stack overflow detection and stack size estimation algorithm [136].

5.2.1. Stack Overflow Detection Algorithm

In this section, we present a dynamic algorithm for detecting stack overflows and estimating thread stack usage. This algorithm on prior work by Park et al. [136]. In order to identify stack overflows in a multithreaded operating system, the following information is required:

1. The *currently active thread* executed by the operating system at a particular point in time.

2. The allocated *thread stack size*, i.e. the total size of the stack memory region for the currently active thread.

3. The amount of *used thread stack* memory at a particular point in time during execution of a thread.

Notably, the same information is required to estimate stack size requirements of executed threads. Figure 5.2 presents an algorithm from prior work for stack overflow detection and stack size estimation which utilizes the aforementioned information. The algorithm exploits the fact that information is stored on a per-function basis on the stack. The algorithm runs *before* each function execution, which is paramount to preemptively detect stack overflows, thereby preventing the system from malfunctioning.

Firstly, the algorithm determines the currently active thread (Line 2) on which the given function is about to be executed. Afterward, in Line 3, the amount of stack space required to execute the function safely is computed. This computation is based on the amount of stack space currently in use and the stack space required by the given function (e.g. memory used for local function variables). The resulting value is then compared to the total thread stack size in Line 5. If the required thread stack size exceeds the total allocated thread stack size, execution of the current function would lead to a stack overflow and a stack overflow handler is invoked in Line 6. Otherwise, the maximum stack usage for the current thread is updated, if it exceeds a previously measured value (Line 7 - Line 8).

Prior work by Park et al. presents an implementation of this algorithm which relies on a modified C compiler and instrumentations of the executed software [136]. In Subsection 5.2.2, we propose a VP-based implementation of this algorithm which allows performing the analysis without any instrumentation or modification of the tested software.

5.2.2. Implementation

In the following, we present an implementation of the outlined stack overflow detection and stack size estimation technique which is tightly integrated with virtual prototyping.

5.2.2.1. Overview

Figure 5.3 shows an overview of our approach. In order to integrate the stack overflow detection and stack size estimation algorithm with a SystemC-based VP, only the ISS has to be modified. The ISS (left to the center of Figure 5.3) is responsible for fetching, decoding, and executing instructions. On each instruction execution, we check if the currently executed instruction—as specified by the program counter register—corresponds to the entry address of a new function. If so, we execute the algorithm from Subsection 5.2.1 before executing the instruction specified by the current program counter.

As per Subsection 5.2.1, the following information needs to be obtained and managed by the VP alongside software execution for an implementation of the algorithm: (1) the currently active thread, (2) the allocated thread stack size, and (3) the amount of thread stack memory in use. Figure 5.3 provides an overview of the components required to extract this information. Extracting the currently active thread (1) and the allocated thread stack size (2) is an operating system specific process. Our architecture provides an abstract operating system support component (left to the center of Figure 5.3) which is responsible for extracting this information. Commonly, metadata information for threads is stored in so-called Thread Control Block (TCB) by the operating system (right to the

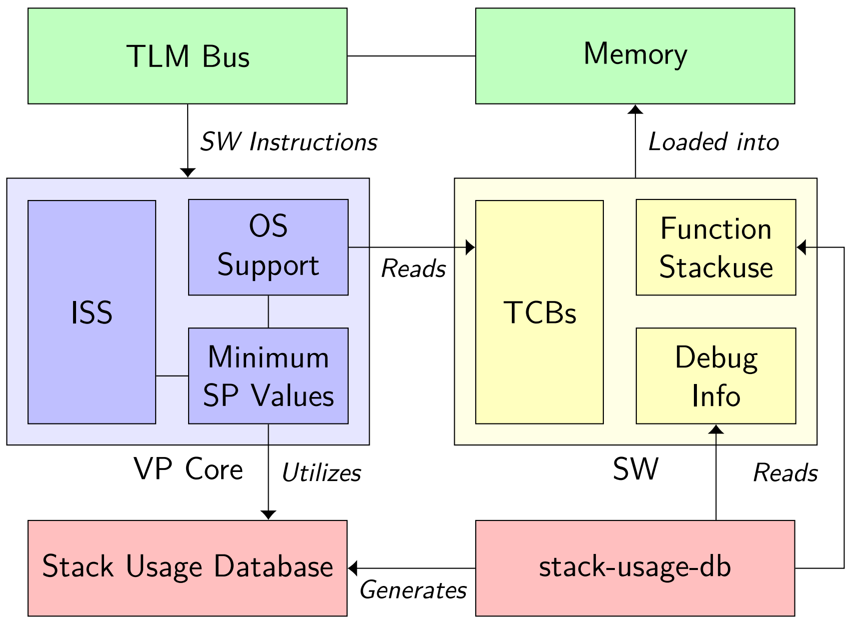


Figure 5.3.: Architectural overview of our proposed approach.

center of Figure 5.3). The operating system support component accesses these TCBs to extract the active thread and the total thread stack size for operating system threads. This component needs to be manually implemented for the specific operating system utilized by the executed software. We present an exemplary operating system support component implementation for RIOT in Subsection 5.2.3.

Regarding the extraction of currently used stack memory (3), recall that the stack is just a memory region where the executed software stores information about active functions (e.g. local variables). From the VP perspective, an access to information on the stack is just a load/store instruction relative to the current position in the stack memory region. Instruction set architectures provide a general purpose register, called the Stack Pointer (SP), to store the current position in the stack memory region. As such, the amount of currently used stack memory can be determined by consulting this register. Assuming the stack grows downward, the maximum stack usage of a given thread can be determined by storing the minimum SP value measured on a per-thread basis (left to the center of Figure 5.3).

On each executed instruction, the VP consults a generated database to check if the instruction address matches the start address of a function. This database is referred to as “Stack Usage Database” at the bottom left of Figure 5.3. The database provides a mapping *Function Address* $\mapsto$ *Function Stack Usage* and is thus used by the VP to predict the total required stack space for each function before executing it, i.e. as done in

Line 3 of Figure 5.2. The stack usage database is generated through a custom tool called `stack-usage-db` using information extracted from the compiled software (bottom right of Figure 5.3). This process is further described in the following section.

5.2.2.2. Stack Usage Database

Modern versions of the GCC compiler toolchain support a command-line flag called `-fstack-usage`. This flag causes the compiler to emit stack usage information for individual functions. In C, the compilation process involves compiling separate translation units into separate object files, these object files are then passed to a linker which generates an executable file from the object files [96, Section 5.1.1]. With the `-fstack-usage` command-line flag activated, GCC outputs a separate file with stack usage information for each compiled translation unit in addition to the object file.

An example stack usage file is shown in Listing 5.3. The file consists of multiple lines, each representing information about a function defined in the associated translation unit. Each line consists of three fields providing different information. The first field states the source file where the function is defined, the line/column number, and the function name. The second field states the stack usage in bytes. The third is a qualifier which further specifies how the function uses the stack. A function may have unbounded stack usage if it uses Variable Length Arrays (VLAs) where the size of an object on the function stack depends on a variable (e.g. a function parameter) [96, Section 6.7.5.2]. Functions with an unbounded stack are not supported by our approach but are automatically identified by our tooling—using the aforementioned qualifier—and cause an error message to be emitted. The experiments we performed with RIOT indicate that VLAs are not widely used in the low-end IoT domain.

The problem with the format shown in Listing 5.3, is that it does not contain any information about text segment addresses of these functions because this information is only available after linking all object files into a binary. However, as per Subsection 5.2.2.1 functions must be identified by their address. For this reason, we wrote a tool—referred to as `stack-usage-db` in Figure 5.3—which merges multiple `-fstack-usage` files into a single stack usage database which is indexed by function text segment addresses. In order to identify the text segment addresses of utilized functions, the tool operates on a linked ELF binary. It iterates over all function symbols defined in the binary and determines the `-fstack-usage` file for a given symbol from DWARF [55] debug information contained in the binary. That is, the DWARF source line information, which describes where a symbol is defined, is compared against the first field of all `-fstack-usage` files generated by the

```
nano-vfprintf.c:392:1:__sfputc_r 0   static
nano-vfprintf.c:403:1:__sfputs_r 32  static
nano-vfprintf.c:348:1:__sprint_r 16  static
stdio.h:503:5:_vfprintf_r        176 static
stdio.h:206:5:vfprintf           0   static
```

Listing 5.3.: Example `-fstack-usage` file generated by GCC.

compiler. If the `-fstack-usage` file for a given symbol was found, the stack usage in bytes for the function represented by this symbol is added to the database. Thereby iteratively creating a mapping *Function Address* $\mapsto$ *Function Stack Usage*. Lastly, the database generated by the `stack-usage-db` tool is passed to the VP on simulation start. This enables the VP to determine whether a new function is being executed (by checking if the database contains a function starting at program counter) and allows determining the stack usage requirements of this function. The source code of the `stack-usage-db` tool is available on GitHub.[5]

5.2.2.3. Operating System Integration

Apart from stack usage information about individual functions, the VP also needs to determine the currently executed thread and its associated stack size. As discussed in Subsection 5.2.2.1, extracting this information requires operating system specific code because multithreaded systems represent information about threads in different ways. However, to implement a scheduler, the operating system will store metadata information for threads in TCBs. TCBs are stored in memory; therefore, it is possible for the VP to extract information about a specific thread by accessing the memory location where this information is stored.

Similarly, operating systems often include symbols to allow a debugger to determine the currently active thread, the offset of information in the TCBs, et cetera. Our approach relies on the operating system to provide such symbols. As part of the operating system support component, we extract the address of these symbols in the executed ELF file using `libdwfl` from `elfutils`[6] directly in the VP. For example, this allows us to determine the memory address of the variable which stores the currently active thread. In the VP context, a SystemC TLM read transaction is then emitted for this address, thereby causing the VP to retrieve the active thread ID from guest memory. Embedded

[5]`https://github.com/agra-uni-bremen/stack-usage-db`
[6]`https://sourceware.org/elfutils/`

operating systems (such as RIOT) store the allocated total stack size in the TCB as well, thus also allowing the VP to access this information via the operating system support component.[7] The next section further describes extraction of information from TCBs using RIOT as an example operating system.

5.2.3. Evaluation

In order to evaluate our technique, we have implemented it on top of `riscv-vp` [85], i.e. the concrete VP on which SymEx-VP is based (see Chapter 3). In total, we had to modify roughly 600 LOC in `riscv-vp`, which shows that a non-intrusive integration of our technique into existing VPs is possible. Similar to Subsection 5.1.3, we then evaluated our implementation by applying it to the RIOT operating system. We choose RIOT for our experiments because multithreading is a core concept of this operating system. Most importantly, RIOT's default network stack (GNRC) implements each network protocol as a separate thread. Different protocol implementations communicate with each other using message passing, a form of interprocess communication. For example, the IPv6 and UDP implementation run in separate threads and the IPv6 thread passes network packets to the UDP thread for further processing [112]. As such, RIOT-based embedded applications consist of multiple threads. Each thread has its own statically allocated thread stack, thus the preallocated stack space has a significant impact on RIOT's memory footprint. In 2018, a minimal RIOT configuration required 3.2 kB of ROM and 2.8 kB of RAM, of which 2.2 kB were thread stack space [9, Section 8]. Presently, RIOT thread sizes are approximated through predefined macros such as `THREAD_STACKSIZE_-DEFAULT`. Our hypothesis regarding stack size estimation is therefore that RIOT threads are overprovisioned in terms of thread stack size, i.e. they do not use the entirety of stack space assigned to them.

In the following, we will describe how we integrated our technique with RIOT (Subsection 5.2.3.1) and discuss stack overflows we encountered in RIOT during this integration (Subsection 5.2.3.2). Furthermore, we will report results on the measured thread stack size in preexisting RIOT test applications and compare our results with the approximated preallocated stack space (Subsection 5.2.3.3). Lastly, we will evaluate the performance impact our proposed technique has on VP execution speed (Subsection 5.2.3.4). The artifacts for this evaluation are available on Zenodo [190].

[7]If this information is not available in the TCB, it can be supplied separately.

5.2.3.1. Integration

As described in Subsection 5.2.2, our approach does not require instrumentations or modifications of the tested software. Instead, we extract information about active threads by reading guest memory from the VP. RIOT already provides dedicated debug symbols which allow a debugger to determine information about active threads. For example, RIOT provides a symbol which allows retrieving an identifier for the currently active thread. This information is already used by debuggers, such as OpenOCD[8], to allow for selective debugging of individual RIOT threads. Based on these symbols, we extract the TCBs for RIOT threads and offsets for information stored inside the TCBs.

In RIOT, all thread stack spaces are disjunct. We infer the currently active thread from the SP value by iterating over all thread stack spaces and checking to which stack space the current SP value belongs. Alternatively, it would also be possible to determine the current thread by ID. However, RIOT uses a dedicated stack for handling interrupts (referred to as Interrupt Service Routine (ISR) stack in the following). Code executed on the ISR stack does not belong to any thread and has no thread ID. As such, identifying the current thread by SP allowed us to allow reasoning about ISR stack usage. This enabled the detection of ISR stack overflows, on which we elaborate in the following.

5.2.3.2. Stack Overflows

As part of our experiments, we uncovered two overflows of RIOT's ISR stack. Both occurred under specific operating system configurations. For debugging purposes, RIOT includes multiple builtin `printf` invocations which are abstracted through a preprocessor `DEBUG` macro and normally disabled. These debug statements can be enabled on a per-file basis. When doing so, the stack space of the associated thread is normally increased by `THREAD_EXTRA_STACKSIZE_PRINTF`. This is necessary as the `printf` family of functions has a comparatively large stack usage. Unfortunately, this approach does not work for the ISR stack since this stack is not allocated in a C file as a static `char` array but instead preallocated in the linker script. As such, the size of the ISR thread stack is not increased when debugging is enabled. Therefore, enabling debug statements in functions executed on the ISR stack causes stack overflows and can lead to hard-to-debug malfunctioning during debugging. We encountered this issue while debugging the RIOT thread creation code from `core/thread.c`. We have also reported this issue to RIOT developers.[9]

[8] `http://openocd.org/`

[9] `https://github.com/RIOT-OS/RIOT/issues/16395`

RIOT also includes a build configuration, called `DEVELHELP`, which enables more helpful error messages but does not allow for verbose debugging of individual files. As an example, the trap handler for the RISC-V architecture prints the value of differs RISC-V CSRs, using `printf`, if `DEVELHELP` is enabled and an unknown trap is encountered. This is useful for easily determining where an unexpected trap occurred. However, since the trap handler is also executed on the ISR stack and the ISR stack is too small to execute `printf` functions, this code path also results in a stack overflow. We encountered this stack overflow as we initially raised a custom trap in the VP when encountering stack overflows. Since this trap is unknown to RIOT, it would cause RIOT to attempt to print the aforementioned debug information which would then result in a stack overflow on the ISR stack and a nested raise of the corresponding trap. We also reported this issue to RIOT developers, one way of fixing it would be switching to a more stack space efficient method for printing CSR values in the trap handler.[10] The fact that we managed to identify two edge cases where a stack overflow occurs in RIOT illustrates the effectiveness of our proposed technique.

5.2.3.3. Stack Size Estimation

In order to evaluate the stack size estimation aspect of our proposed technique, we measured the maximum stack usage for preexisting test cases for RIOT's network stack GNRC. The results are shown in Table 5.3. Each application starts multiple threads and was executed until a predefined cancellation point was reached (e.g. `TestRunner_end`). For each thread in each test case, Table 5.3 shows the measured maximum stack usage and the configured stack size in bytes. Lastly, the percentage of configured stack space that was actually used by the test case is shown in the last column of Table 5.3.

The majority of executed test cases use less than 50 % of the configured stack space. This confirms our initial hypothesis that RIOT threads are overprovisioned in terms of stack size (see Subsection 5.2.3). It is also noticeable that stack sizes are often reused and not tailored to a specific application. Most notably, all executed test cases use the default main thread stack size of 1024 B. Please note though that the executed test cases are not specifically designed to yield the worst case stack usage. As such, measurements from Table 5.3 only indicate the maximum stack usage measured, but not necessarily the worst possible stack usage. Nonetheless, they serve as a good indicator and may help developers in iteratively optimizing the stack sizes of their application.

[10]`https://github.com/RIOT-OS/RIOT/issues/16448`

Table 5.3.: Stack size estimated for RIOT's GNRC test cases.

		Stack Size		
Test Case	Thread	Maximum	Configured	Used
`gnrc_ipv6_nib`	`ISR`	128 B	512 B	25 %
	`idle`	76 B	256 B	29.69 %
	`ipv6`	544 B	1024 B	53.12 %
	`main`	716 B	1280 B	55.94 %
	`mockup_eth`	516 B	1024 B	50.39 %
`gnrc_ndp`	`ISR`	128 B	512 B	25 %
	`idle`	72 B	256 B	28.12 %
	`main`	456 B	1280 B	35.62 %
	`test-netif`	440 B	1024 B	42.97 %
`gnrc_rpl_p2p`	`ISR`	96 B	512 B	18.75 %
	`idle`	80 B	256 B	31.25 %
	`ipv6`	228 B	1024 B	22.27 %
	`main`	304 B	1280 B	23.75 %
`gnrc_sock_udp`	`ISR`	176 B	512 B	34.38 %
	`idle`	80 B	256 B	31.25 %
	`ipv6`	388 B	1024 B	37.89 %
	`main`	576 B	1280 B	45 %
	`udp`	304 B	1024 B	29.69 %

Table 5.4.: Benchmarks results for `tests/bench_runtime_coreapis`.

Benchmark	Modified VP	Baseline	Slowdown
`nop loop`	0.71 s	0.33 s	53.52 %
`mutex_init`	0.0 s	0.0 s	0 %
`mutex lock/unlock`	14.8 s	6.7 s	54.73 %
`thread_flags_set`	7.22 s	3.3 s	54.29 %
`thread_flags_clear`	3.98 s	1.61 s	59.55 %
`thread flags set/wait any`	19.96 s	8.5 s	57.41 %
`thread flags set/wait all`	17.37 s	7.33 s	57.8 %
`thread flags set/wait one`	21.31 s	8.91 s	58.19 %
`msg_try_receive`	10.75 s	4.26 s	60.37 %
`msg_avail`	3.03 s	0.99 s	67.33 %
Average	-	-	52.32 %

5.2.3.4. Performance Impact

As discussed in Subsection 5.2.2, our approach relies on sanity checks performed during the execution of RISC-V instructions and thus has an impact on execution performance. We measured this impact by executing RIOT benchmarks specifically designed to gather performance statistics.

The utilized benchmark performs multiple consecutive invocations of different functions from the RIOT API. In Table 5.4 we compare our implementation against the original unmodified `riscv-vp` version as presented in prior work [85]. All tests have been performed on an Intel i7-8565U system running Alpine Linux. The first column in Table 5.4 (Benchmarks) shows the executed benchmark function, the second the time it took to execute it with our implementation (Modified VP), and the third the execution time with the original `riscv-vp` (Baseline). The fourth column (Slowdown) displays the relative slowdown caused by our implemented stack overflow detection and stack size estimation technique. On average, execution is slowed down by 52.32 % through our employed technique.

We believe this to be an acceptable overhead during development. Currently, our implementation performs sanity checks for each executed instruction to implement the algorithm from Subsection 5.2.1. The performance impact may be reduced by only performing these checks after jump instructions.

5.2.4. Related Work

Prior work has already proposed a variety of techniques to prevent stack overflows and/or reduce stack usage on embedded systems. One solution for reducing stack usage is to resize thread stacks dynamically as needed [20, 103, 15]. In this regard, Biswas et al. propose adding additional sanity checks to the compiled software to detect stack overflows and resize the stack segment if an overflow is imminent [20]. Similarly, Kim et al. measure stack usage periodically at runtime and perform stack reallocations if needed [103]. Behren et al. present an operating system which adopts a dynamic stack allocation technique [15]. Unfortunately, these techniques impact runtime behavior and can also cause memory fragmentation, both of which we believe to be undesirable, especially on constrained devices which provide real-time guarantees.

For this reason, a different branch of research focuses on determining worst case stack usage prior to software deployment. A popular approach for doing so is static analysis [149, 101, 30]. Regehr et al. and Brylow et al. present such an approach which specifically focuses on interrupt-driven embedded software [149, 30]. However, static approaches cannot handle recursive functions and indirect function calls. This problem is mitigated in prior work by requiring programmers to provide annotations for loop bounds and indirect calls. For example, prior work by Kästner et al. requires programmers to manually add annotations in the formal AIS language [101]. Unfortunately, manual effort makes it more laborious to employ such techniques.

Lastly, different dynamic testing approaches for finding stack overflows have been proposed [136, 211, 148]. Regehr uses random testing to determine worst case stack usage and compares results with the aforementioned static approach by the same author [148]. Prior work by Zhang et al. relies on memory protection mechanisms provided by the processor [211]. Related work by Park et al. uses a modified C compiler to add sanity checks to the preamble of each compiled function to detect stack overflows and estimate worst case stack size [136]. This hinders adaption of such approaches. In order to ease employment in the constrained embedded domain, we believe it to be desirable to detect stack overflows during normal testing already performed today with VPs in early stages of software development. Contrary to prior dynamic testing approaches, our approach enables an analysis that requires no modification or instrumentation of the executed software.

5.2.5. Discussion and Future Work

The evaluation demonstrates that our approach is capable of uncovering stack overflows in real-word software for constrained devices. As such, we have demonstrated that the technique is a suitable path analyzer for reasoning about stack overflows and estimating stack sizes. So far, we have only implemented the proposed algorithm on top of `riscv-vp`, we have not yet integrated it with symbolic execution and SymEx-VP (see Chapter 3). However, we believe that such an integration is primarily an engineering effort. Nonetheless, such an integration would be beneficial since symbolic execution allows us to enumerate (ideally all) reachable programs paths automatically. In regard to constrained devices, this is especially interesting when considering our proposed stack size estimation technique. Due to memory limitations, the state space on constrained devices is often smaller than on conventional ones, potentially even allowing for a complete analysis using symbolic execution in this domain [54]. If a complete analysis is possible, this could allow offering guarantees regarding the estimated thread stack size.

5.2.6. Conclusion

We presented a stack overflow detection and stack size estimation technique for multi-threaded operating systems which we implemented using VPs in a way that does not require software instrumentation (Subsection 5.2.2). This allows finding stack overflows and gives programmers an estimate regarding thread stack usage during early software development. Compared to the work presented in Section 5.1, the proposed technique only detects a particular kind of memory corruption (stack overflows) but does not require software modifications. By avoiding software instrumentation, we can ensure that the observed behavior does not change when deploying the software in a production environment. Our implementation is specifically tailored to constrained devices where stack overflows would normally go undetected due to the lack of memory protections. In this regard, our technique enabled us to find two previously unknown stack overflows in the low-end IoT operating system RIOT, which we reported to RIOT developers (Subsection 5.2.3.2). Additionally, preliminary results obtained using our stack size estimation technique indicate that existing RIOT application potentially overestimate thread stack sizes, thereby wasting memory (Subsection 5.2.3.3). We intend to further improve our proposed technique in future work by combining it with symbolic execution and SymEx-VP. Our current implementation is freely available on GitHub.[11]

[11]`https://github.com/agra-uni-bremen/fdl21-stackuse-vp`

5.3. Reliable Memory Safety using Safe C Dialects

In Section 5.1 and Section 5.2, we have presented path analyzers for detecting memory corruptions. These analyzers are specifically tailored to the embedded domain and intended to be used during automated software testing with symbolic execution. Unfortunately, for complex whole system analysis (e.g. testing the network stack of the RIOT operating system), symbolic execution is commonly not complete due to path explosion issues. Therefore, we cannot guarantee the absence of memory corruptions and instead focus on bug hunting. Nonetheless, due to the disastrous consequences of memory corruptions, we want to ensure that we include runtime checks in our firmware to prevent memory corruptions from occurring in a production environment. Therefore, in this section, instead of focusing on the detection of memory corruptions during automated software testing, we intend to guarantee their absence (i.e. achieve spatial memory safety). As discussed in Chapter 5, a promising direction in this regard is the utilization of programming language which achieve spatial memory safety through runtime bounds checks. In the embedded domain, new safe languages like Rust[12] are becoming increasingly popular for this reason [114].

However, rewriting a large existing code base (like the RIOT operating system) in an entirely new programming language like Rust is a major effort. For this reason, there is a vast body of prior work on so-called *safe C dialects*. These dialects attempt to extend the C programming language with facilities for writing memory-safe C code [65, 47, 99]. Contrary to new languages like Rust, they do not require full rewrites of an existing code base but instead rely on annotations (e.g. to specify pointer bounds) and minor local conversions. A recent advancement in this domain is Checked C [65] which is under ongoing development by Microsoft and provides facilities for writing memory-safe C code. Compared to other safe C dialects, Checked C is a fully backward compatible extension of the standard C programming language. In order to distinguish the two, we refer to the latter as *legacy C* in the following. Due to backwards compatibility, conversion from legacy C to Checked C can occur incrementally and on demand. Moreover, incremental conversions enable developers to selectively convert critical parts to Checked C first, thereby improving applicability to larger programs. While Checked C developers are actively working on tooling for partially automating the conversion process, this tooling is intended for “simple cases” and programmers still need to annotate bounds manually [156, Section 2]. As such, the conversion process relies on manual effort, which is

[12] `https://www.rust-lang.org/`

susceptible to errors and thus requires cross-validations. We refer to errors introduced during the conversion process as conversion bugs. The majority of these accidentally introduced conversion bugs are hopefully discovered through existing unit tests. However, especially when considering bugs caused by incorrectly specified pointer bounds, preexisting test cases may not be sufficient to check all relevant edge cases.

In order to uncover conversion bugs, we propose to combine Checked C with symbolic execution to obtain an effective methodology for finding execution paths containing conversion bugs. Experiments conducted with the RIOT operating system in this regard show that Checked C also allows us to make previously silent memory corruptions observable. Hence, as a useful by-product, we can also facilitate the additional runtime bounds checks inserted by the Checked C compiler as a path analyzer for discovering spatial violations. Apart from conversion bugs, we were therefore able to discover four previously unknown memory corruptions in RIOT's network stack as part of conducted experiments.

5.3.1. Background Information on Checked C

Checked C is a backwards-compatible extension of the C programming language with a focus on extending C with facilities for writing memory safe C code. As discussed in Subsection 5.1.1.1, memory safety has two aspects: temporal safety and spatial safety [65, Section 1]. Similar to HardBound, Checked C presently only focuses on spatial safety. Spatial memory safety is achieved in Checked C through the addition of new pointer types (referred to as *checked pointers* in the following). From these new pointer types, pointer arithmetic is only allowed on values of a specific checked pointer type, this type is called `_Array_ptr` in Checked C. Pointers of this type must have associated pointer bounds information. Bounds are specified through so-called *bounds expressions* which describe the memory range that can be accessed through a given variable [65].

Generally speaking, converting existing legacy C code to Checked C involves changing raw C pointers to checked pointer types and annotating pointer bounds through bounds expressions where necessary. Fully converted code parts can be marked as *checked regions*, these regions can be held "blameless for any spatial safety violation" [65, Section 1], i.e. no spatial safety violations can arise due to code marked as a checked region. Since Checked C extends legacy C with new syntactic constructs, a custom compiler is required. We utilize the LLVM-based reference implementation.[13]

[13]`https://github.com/microsoft/checkedc-clang`

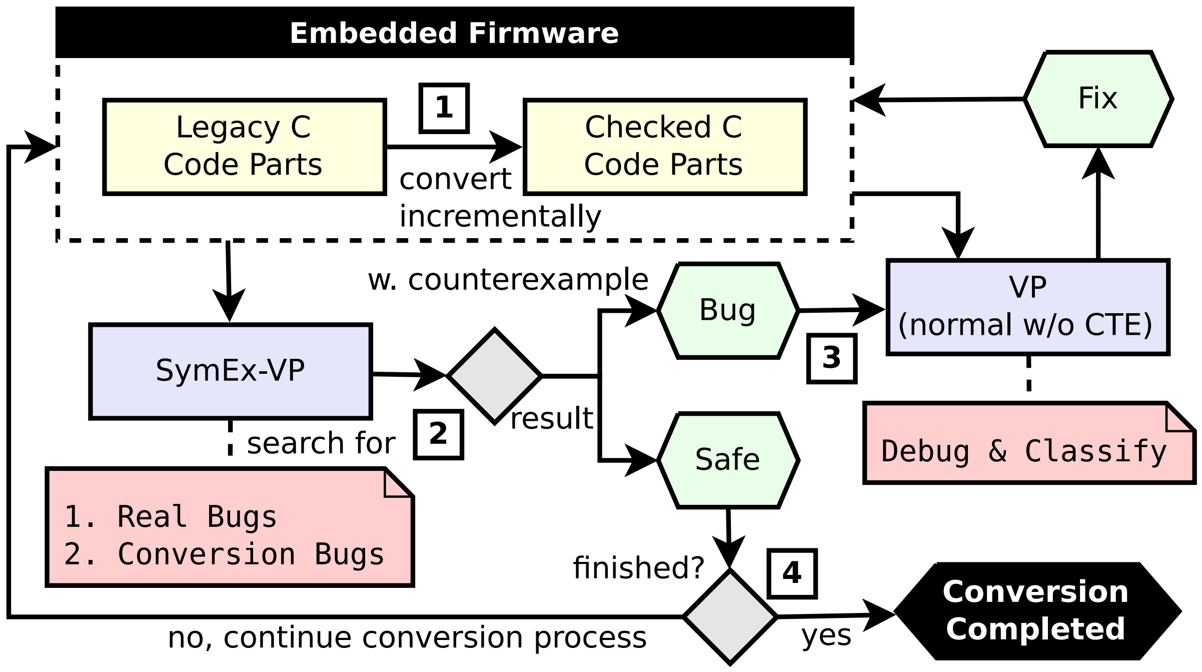


Figure 5.4.: Proposed methodology for using symbolic execution to ease conversions.

5.3.2. Methodology

This section presents our methodology on combining Checked C with symbolic execution to attain more reliable spatial memory safety for embedded firmware. Figure 5.4 provides an overview of the methodology. The starting point is an embedded firmware written in legacy C that is converted to Checked C (position $\boxed{1}$ in Figure 5.4). We expect this process to occur incrementally based on logical firmware modules, as a full conversion at once is deemed difficult and may not be achievable. For this reason, the embedded firmware will typically consist of both Checked C and legacy C code parts. After each increment, the newly converted modules are tested through symbolic execution using our SymEx-VP symbolic execution engine (position $\boxed{2}$ in Figure 5.4). As described in Chapter 3, SymEx-VP follows a standard symbolic execution methodology, it operates on the binary level and searches for inputs that will cause an execution error (e.g. failed Checked C bounds checks in our case). More specifically, this approach enables us to discover two kinds of bugs in the (partially) converted code base:

1. Real spatial memory safety violations which were already present in the unmodified original legacy C code base and can result in—potentially exploitable—undefined behavior without Checked C.

```c
bool
parse(char *input, size_t len)
{
 char *end = input + len;
 char *buf = input;

 // Advance buf til seperator is found
 while (buf < end) {
  if (*buf == '.' || *buf == ',')
   break;
  buf++;
 }
 if (buf == end)
  return false;

 // Extract key and value relative to buf
 char *key = input;
 char *value = buf+1;

 // Consult seperator for further parsing
 if (*(value - 1) == '.')
  return parse_period(/* ... */);
 else
  return parse_comma(/* ... */);
}
```

```c
bool
parse(_Array_ptr<char> input : count(len), size_t len)
_Checked {
 _Array_ptr<char> end = input + len;
 _Array_ptr<char> buf : bounds(input, end) = input;

 // Advance buf til seperator is found
 while (buf < end) {
  if (*buf == '.' || *buf == ',')
   break;
  buf++;
 }
 if (buf == end)
  return false;

 // Extract key and value relative to buf
 _Array_ptr<char> key : bounds(input, buf) = input;
 _Array_ptr<char> value : bounds(buf+1, end) = buf+1;

 // Consult seperator for further parsing
 if (*(value - 1) == '.')
  return parse_period(/* ... */);
 else
  return parse_comma(/* ... */);
}
```

Listing 5.4.: Implementation of an input parser in legacy C (left) and Checked C (right).

2. Conversion bugs which were introduced while converting the legacy C code base to Checked C, i.e. bugs which result in crashes that cannot be reproduced with the original unmodified legacy C code base.

The latter are closely related to Checked C bounds expression which are used to specify pointer bounds to achieve spatial memory safety in Checked C. For each bug, SymEx-VP returns a counterexample, i.e. a test case to reproduce the bug (position $\boxed{3}$ in Figure 5.4). As previously demonstrated in Subsection 3.1.4.2, SymEx-VP provides an extensive debugging interface which can be used to inspect generated test cases. This allows developers to classify the bug accordingly and provide a fix for the embedded firmware. The conversion process continues incrementally until no more bugs are found and all required firmware modules have been converted (position $\boxed{4}$ in Figure 5.4). In the following, we present an example to illustrate the conversion process and conversion bugs (Subsection 5.3.2.1). Afterward, we provide a classification of conversion bugs (Subsection 5.3.2.2).

5.3.2.1. Conversion Bug Example

The distinction between real spatial memory safety violations and conversion bugs is best illustrated using an example. We believe spatial memory safety violations (e.g. buffer overflows) to be well understood and described in existing literature [181]. Therefore, the following example will focus on Checked C conversion bugs, it will also illustrate modifications required to convert existing legacy C code to Checked C. We must stress that the example is kept simple for clarity; thus, the path leading to the included conversion bug would be discovered by a proper unit test. However, similar bugs may arise in paths that are difficult to reach under specific conditions and thus not covered by existing unit tests.

Consider a parser for an input of the form `foo.bar` or `foo,bar` where `foo` is some kind of key and `bar` is some kind of value. An exemplary implementation of such a parser is presented in Listing 5.4. The figure contains both, the original legacy C implementation (on the left-hand side of Listing 5.4) and a version converted to Checked C (on the right-hand side). The two versions are similar and use the same line numbers but the Checked C version uses checked pointer types with bounds expressions for Checked C `_Array_ptr` types on which pointer arithmetic is performed. Furthermore, the Checked C function is marked as a checked region in Line 3 using the `_Checked` keyword, i.e. it can be held "blameless for any spatial safety violation" [65, Section 1]. Both implementations start off by skipping over given input characters until a period or comma character is encountered (Line 8 - Line 12). If no such character is found, `false` is returned (Line 13 - Line 14). Afterward, `key` (Line 17) and `value` (Line 18) variables are declared, the Checked C implementation constrains the bounds of these variables according to the parser format. Lastly, the separation character is again consulted (relative to the `value` variable) in Line 21 to determine whether the extracted value should be further processed with the `parse_period` or the `parse_comma` function.

The legacy C code in Listing 5.4 does not contain any spatial memory safety violations. However, during the conversion to Checked C a conversion bug was introduced: the code in Line 21 for accessing the separation character causes a bounds violation as `value - 1` is one character outside of the bounds specified for the `value` variable. Reconsider the initial input example `foo.bar`, here the variable `key` would be constrained to the first three characters while the variable `value` would be constrained to the last three characters. As such, accessing the period character as "the character before the first `value` character" (as done in Line 21) constitutes an out-of-bounds access. Fortunately, this access is still within the bounds of the `input` buffer and thus does not constitute a memory safety violation in the legacy C implementation. However, in the Checked C implementation, the access will result in program termination due to a failing bounds check. This failing bounds check can be fixed either by widening the pointer bounds of the `value` variable or by accessing the separation character as `*buf` in Line 21.

Even though the example illustrates that converting existing legacy C code to Checked C is comparatively straightforward, it also demonstrates that subtle bugs can be introduced during the conversion process. These bugs may even occur in lines which have not been modified and are therefore easy to miss during manual code review. If undetected, such bugs impact reliability in a production environment by causing crashes. In the worst case, such crashes can allow for a denial-of-service attack which impacts the availability of the system. In the following, we further discuss different kinds of conversion bugs.

5.3.2.2. Conversion Bug Classification

We distinguish the following classes of conversion bugs:

1. *Narrowing bugs* can occur when narrowing pointer bounds. They occur because either (a) a pointer bound was set too narrow, or (b) existing code has not been properly adjusted to respect the newly introduced pointer bounds (the latter was demonstrated by the example in Subsection 5.3.2.1).

2. *Widening bugs* occur when pointer bounds have been set too wide. The Checked C compiler already integrates static analysis techniques to detect such bugs. Using so-called subsumption checks, Checked C ensures the correctness of specified pointer bounds. These checks allow assignments to narrow down—but not to widen—pointer bounds, thereby preventing widening bugs, but not narrowing bugs [65, Section 4].

3. *Functional bugs* refer to bugs that change the functional behavior and are not detected by Checked C (because they have no impact on spatial memory safety). Functional bugs can occur when code is rewritten to handle present limitations of Checked C, e.g. the same variable cannot have different bounds at different points in the program [65, Section 3.7].

In terms of conversion bugs, we focus on the detection of narrowing bugs. Narrowing is explicitly encouraged by Checked C to allow programmers to divide and constrain accessible data as they desire. As an example, the `key` variable in Listing 5.4 narrows the bounds of the `input` variable. Detection of narrowing bugs ultimately concerns itself with finding a path where an access, which violates the defined pointer bounds, is performed. Contrary to widening bugs, this is difficult to detect via static analysis, as it requires reasoning about performed accesses, not about the bounds expressions themselves.

An undetected narrowing bug will cause a bounds check failure at runtime and thus impact reliability of the embedded firmware. In the worst case, an undetected narrowing bug can be exploited as part of a denial-of-service attack which impacts the availability of the system. Depending on the enclosing product of the embedded system, this can have potentially disastrous consequences. Contrary to spatial memory safety violations, narrowing bugs are introduced during the conversion process and cannot occur in the original firmware. By combining Checked C with symbolic execution, we can detect both spatial violations and narrowing bugs. Both are observable during symbolic execution through failing Checked C bounds checks. We further distinguish and demonstrate these two use cases in Subsection 5.3.3. We leave the detection of functional bugs for future work, but we believe that symbolic execution is a suitable foundation for this use case.

5.3.3. Evaluation

Similar to the prior evaluations conducted in Subsection 5.1.3 and Subsection 5.2.3, we evaluate our proposed methodology by applying it to the RIOT operating system. Apart from its popularity, we also choose the RIOT as a test target because it employs a modular software architecture. Therefore, RIOT's architecture allows for an incremental conversion of the existing source code to Checked C on a per-module basis. This is beneficial for an application of our methodology. In the following, we provide more details on our test setup (Subsection 5.3.3.1) and present our obtained results (Subsection 5.3.3.2).

5.3.3.1. Setup

In our evaluation, we consider the network stack of RIOT, which is a crucial and central component of every IoT system. In particular, we focus on core network modules, including IPv6, ICMPv6, DNS, and utility libraries such as URI parsers. Following our methodology, we incrementally converted these modules to Checked C. In accordance with prior research, our analysis focused on input handling routines of the network stack, which receive input from a network connection, since these are deemed most vulnerable to potentially exploitable spatial violations such as buffer overflows [159]. In total, we incrementally converted 53 functions in 6 different RIOT modules.

In the spirit of Subsection 3.1.5, we employed a two-step methodology for testing the incrementally converted code using SymEx-VP. That is, we conducted both unit and integration tests. As a first step, we created specialized unit tests which target specific functions and modules. For unit testing, we therefore created a respective test harness (which is a piece of C code) that calls the software under test with symbolic input values. As a second step, we utilized existing RIOT example applications for more extensive integration testing by introducing symbolic input values directly into the network stack using the previously presented SLIP network interface. Recall that SLIP is a network protocol for the transmission of IP packets over a serial line [155]. In our setup, we pack symbolic data into an IPv6 packet on the SymEx-VP side, encapsulate it in a SLIP frame, and pass it through a UART into the RIOT SLIP network driver. The driver processes the SLIP frame and forwards it to the network stack. Since the network stack awaits input indefinitely, we terminate execution after one packet has been processed.

We register a custom abort handler in RIOT to notify SymEx-VP about detected errors. The abort handler is called if a Checked C bounds check fails at runtime, but also if existing RIOT assertions fail. Based on the concrete input emitted by SymEx-VP, we can create a test case to reproduce and debug the source of an error, and thereby classify the bug kind (i.e. real spatial memory bug, Checked C conversion bug introduced by our conversion process, or other logic bug). Please note, that we target the RISC-V architecture in this evaluation (and hence the low-level routines in the RIOT code, such as context switching, use RISC-V specific code), but the higher-level network stack routines themselves are written in platform independent C code. As RIOT supports executing high-level application code as a native `x86` Linux binary, we can utilize conventional development tools for detecting spatial violations which are not available on bare-metal RISC-V (e.g. Valgrind [133] or AddressSanitizer [163]) to classify discovered bugs. This is achieved by evaluating whether a generated concrete input also results in the detection of a spatial or logic bug using these tools with an unmodified version of RIOT. If not, this serves as an indication that the bug was introduced during the conversion.

5.3.3.2. Results

Table 5.5 lists all errors that we have found in network-related RIOT modules. The table has five columns: (1) the bug identifier (column: ID), (2) the RIOT component where the bug has been found (column: Component), (3) the type of test (UNIT test or using the SLIP interface) that led to the detection, (4) the number of (symbolic) execution paths explored by SymEx-VP until the bug was found (column: #Path), and (5) the overall

Table 5.5.: Spatial violations (M1 - M2), real logic bugs (L1 - L2), and Checked C conversion bugs (C1 - C3) that our approach found in different RIOT modules.

ID	Component	Test	#Paths	Time
M1	`uhcp`	UNIT	282	64.50 s
M2	`sock_dns`	UNIT	5	1.28 s
L1	`gnrc_nib`	SLIP	75	52.37 s
L2	`gnrc_netif`	SLIP	67	47.95 s
C1	`gnrc_icmpv6`	SLIP	66	45.31 s
C2	`uri_parser`	UNIT	196	40.00 s
C3	`clif`	UNIT	17	3.84 s

execution time in seconds required to find the bug (column: Time). All experiments have been conducted on a Linux system with an Intel Xeon Gold 6240 processor.

In total, we found 7 unique errors, four of these being real bugs, which are further classified into spatial memory (M1-M2) and logic bugs (L1-L2), and three being conversion bugs (C1-C3). Logic bugs constitute failing C assertions and can also be detected without Checked C, all other bugs are specific to Checked C. All real bugs (M1, M2, L1, L2) that we detected have been previously unknown and have already been confirmed by the RIOT developers. This demonstrates the effectiveness of our combined testing approach. Moreover, to trigger certain bugs, specific input parameters are required which we deem difficult to discover using other techniques. Usage of SymEx-VP has also been beneficial in supporting the Checked C conversion procedure. Despite being cautious during the conversion process, we discovered three conversion flaws with SymEx-VP. We would like to point out that, due to the manual conversion process, such kinds of bugs can be easily added accidentally. Undetected, they would result in a runtime check failure and thus abort the embedded software application, thereby impacting reliability and—in the worst case—allow for a denial-of-service. Table 5.6 provides a more detailed description for each of the 7 bugs, including references to the RIOT bug tracker.[14]

[14] `https://github.com/RIOT-OS/RIOT/issues`

Table 5.6.: Summary of all real and conversion bugs discovered using our approach.

Type	Bug Description
Real Memory Bug	**M1 [#15353]:** Buffer overflow during parsing of the IPv6 network prefix. The `uhcp` module contained a `memset` invocation with an incorrect length parameter, resulting in a stack-based buffer overflow. **M2 [#15345]:** A bounds check performed in the RIOT DNS implementation was incorrect. This allowed for a two byte out of bounds buffer access during DNS response parsing.
Real Logic Bug	**L1 [#15171]:** The RIOT Neighbour Information Base (NIB) implementation for IPv6 contained a failing assertion. This assertion could only be reached when using a SLIP network interface. **L2 [#15221]:** Failure to release a mutex on return. The RIOT `gnrc_netif` module, which provides an abstraction for network interfaces, contained a path where a mutex was locked but not unlocked on return. Reaching this specific path resulted in a deadlock (detected via a timeout mechanism).
Conversion Bug	**C1:** The RIOT ICMPv6 implementation parses protocol headers by casting packed structs on pointers. Most of these casts require a dynamic Checked C bounds check. In one instance, RIOT performed a bounds check after casting the pointer, thereby resulting in a failing Checked C bounds check. **C2:** RIOT provides a non-destructive parser for URI references. The parser splits the URL into different parts (scheme, host, port, etc.) but in one instance accesses data outside the host part. However, this access was still within the bounds of the underlying URL buffer and did thus not constitute a real memory safety violation. This is conceptually similar to the issue described in Subsection 5.3.2.1. **C3:** The RIOT `clif` module provides a parser which increments a `uint8_t` pointer contiguously. This causes the pointer bounds to be narrowed on each increment, at one point the previous pointer value was accessed after performing an increment thus resulting in a spurious bounds violation.

5.3.4. Related Work

We believe Checked C to be one of the most recent advancements in the safe C dialect domain. Naturally, prior work has also presented alternative safe C dialects, such as Cyclone [99] and Deputy [47]. Contrary to Checked C, these dialects are either not fully compatible with legacy C or no longer in active development. Apart from safe C dialects, existing literature also considers alternative techniques for achieving memory safety in legacy C code, e.g. [131, 60]. An initial evaluation done by Checked C developers indicates that Checked C has a runtime overhead of 8.6 % and an executable size overhead of 7.4 % on average [65]. We consider these overheads to be low, compared to other techniques proposed in prior work, which is why we believe Checked C to be a promising technique for achieving spatial memory safety in the embedded domain.

In terms of application to existing systems, Checked C itself has previously primarily been applied to conventional operating systems, such as FreeBSD [63]. Its application to the embedded domain is presently limited, however, we believe Checked C to be a promising candidate for this domain as—due to the lack of other protection mechanisms—embedded firmware benefits immensely from language-based safety and security techniques. This hypothesis also confirmed by previous work on the application of prior safe C dialects to other embedded operating systems. In this regard, prior work by Paul et al. employs Deputy in conjunction with the Contiki operating system [137]. Similarly, Cooprider et al. employ Deputy for securing the TinyOS operating system [48]. This prior work illustrates the benefits of using safe C dialects for embedded firmware. However, we consider these approaches complementary to our own since they do not address conversion bugs. To the best of our knowledge, we are unaware of related work which utilizes other software validation techniques in combination with Checked C or any other safe C dialect to improve reliability of converted code.

5.3.5. Discussion and Future Work

Our experiments demonstrate the effectiveness and potential of our proposed methodology for integrating the Checked C conversion process with symbolic execution to achieve reliable spatial memory safety. Nonetheless, there is still room for improvement, which we discuss in the following. Due to state explosion issues, symbolic execution is generally unable to guarantee the absence of errors because the complete state space is commonly not covered. Therefore, it is rather a bug hunting technique. While our experiments demonstrated its capabilities in this regard, we cannot prove the complete absence of failing checked C bounds checks. Checked C itself avoids spatial violations leading to potentially exploitable undefined behavior, but undiscovered reachable failing bounds checks can still impact reliability. In order to further improve our technique, we plant to investigate whether it is possible to achieve completeness by testing small components (e.g. individual functions) in isolation.

In this regard, another interesting research direction is to optimize the runtime checks inserted by the Checked C compiler to reduce code size and lessen the runtime performance impact imposed by Checked C. The reference compiler for Checked C already employs static analysis techniques to prove spatial memory safety of certain operations at compile-time and thus avoid generation of runtime checks [65, Section 3.3]. We envision to utilize symbolic execution (and potentially other formal techniques) to prove that certain runtime checks are not necessary. An incremental approach that starts with isolated functions and combines the results in a compositional way seems promising. Moreover, it would be interesting to investigate the strengthening of the static analysis employed by the Checked C compiler with guarantees on the enumerated paths provided via symbolic execution techniques.

Yet another important direction is to investigate dedicated techniques for detection of functional bugs which may be introduced by the Checked C conversion process. We believe that symbolic execution is a suitable foundation to develop effective bug hunting techniques for finding such bugs efficiently as well. This can be complemented with complete proof techniques to enable equivalence proofs of the converted Checked C software with the legacy C software. While prior work has already utilized symbolic execution for the purpose of equivalence checking, limitations with regard to scalability still remain [147]. An incremental approach, that performs the equivalence proofs in a compositional way module by module (following the incremental Checked C conversion process), might be a viable solution to tackle this problem.

5.3.6. Conclusion

In this section, we proposed to combine Checked C with symbolic execution to attain more reliable spatial memory safety for embedded firmware. We employ symbolic execution to safeguard the incremental conversion process from legacy C to Checked C. Beside conversion bugs, our approach enables detection of spatial memory bugs which have been present in the original embedded firmware. In comparison to our prior work presented in Section 5.1 and Section 5.2, employment of Checked C is not only useful for testing purposes but also offers protection against memory corruptions in a production environment. The effectiveness of our approach was demonstrated by applying it to the network stack of the RIOT operating system. We found four previously unknown bugs in the RIOT network stack, which have been confirmed and fixed by RIOT developers, and three conversion bugs added accidentally that otherwise would have caused a spurious bounds check failure at runtime.

Chapter 6.

Input Generation Heuristics for Applications in the IoT

In prior chapters, we have developed an accurate symbolic execution approach for embedded firmware and have described parts of its implementation including handling of environment models (Chapter 3), binary code instructions (Chapter 4), and path analyzers (Chapter 5). In these prior chapters, we have conducted experiments to evaluate the corresponding contributions. However, we have not yet concerned ourselves with scalability issues of symbolic execution (path explosion) which prior work refers to as the “biggest challenge facing symbolic execution” [33, p. 87]. In order to mitigate path explosion issues, domain-specific heuristics and optimizations are needed. In this chapter, we concern ourselves with conducting experiments and providing heuristics for a particular kind of embedded firmware: IoT applications. IoT applications collect information about their surrounding environment and exchange this information with other systems to achieve a common goal [121, Definition 1.3]. For information exchange, IoT applications utilize IP-based network protocols which were specifically designed for the constraints of devices used in this domain (e.g. MQTT-SN [176]) [24].

Due to their complexity, implementations of these network protocols provide a large attack vector. Unfortunately, it is challenging to test these implementations using dynamic software testing techniques such as fuzzing [139, 132] or symbolic execution [174, 173, 6]. Due to their large state space it is infeasible to explore the entirety of the implementation, instead automated dynamic software tests are often performed with a fixed time budget. Therefore, it is vital to ensure that the most relevant execution paths are explored first [33, p. 87]. Otherwise, critical errors may be missed within the allocated time budget during automated software testing. For this reason, domain-specific search heuristics and optimizations are needed. When testing network protocol implementations, it must be taken into account that the input space is regulated by the protocol specification. That is, the majority of inputs are rejected early on without performing any interesting input processing. This is especially challenging when considering stateful protocols (such as MQTT-SN) where valid input depends on the current state of the protocol state machine (i.e. prior messages).

Prior work on fuzzing attempts to address this issue through the use of protocol specifications based on which input is generated [12, 107, 134]. Unfortunately, creating such protocol specifications can be a cumbersome manual process; therefore, existing work also concerns itself with the automatic creation of such specifications [46]. In this section, we explore a different direction by attempting to ease the creation of protocol specifications through the utilization of symbolic execution. Contrary to fuzzing, symbolic execution takes the program structure into account, this enables us to only partially specify the protocol's input format by treating unspecified parts as unconstrained symbolic values. We then leverage an SMT solver—used in symbolic execution for formal reasoning—to automatically fill in the leftover gaps based on extracted program constraints. To this end, we present an EDSL for the creation of partial protocol input format specifications in Section 6.1. In Section 6.2, we extend this EDSL to also allow the specification of the protocol state machine, thereby enabling verification engineers to express changes of the protocol input space based on the current protocol state. We evaluate this approach by conducting experiments with stateful network protocol implementations (such as MQTT-SN) provided by the Zephyr and RIOT operating systems. As part of the conducted experiments, we uncovered three previously unknown bugs in stateful network protocol implementations provided by the RIOT operating system. Lastly, we concern ourselves with the process of refining creating protocol specifications. Presently, creating a protocol specification is a manual process and therefore susceptible to errors. Specifically, if the specification is too narrow, bugs in the tested software may be missed; if it is too broad, then interesting deeper parts of the implementation may not be reached and—again—bugs may be missed. Therefore, in Section 6.3, we propose visualizing the conducted symbolic analysis to enable incremental refinements of created protocol specifications.

6.1. Input Specification Language for Message Formats

Prior work considers the security of a computer system to be "defined by what computations can and cannot occur in it under all possible inputs" [159, p. 22]. In the case of IoT applications, the input space for a network protocol implementation is given by the protocol specification. In order to optimize input generation for this domain, we need to take the protocol specification and its defined input space into account. A protocol specification covers different aspects, regarding the input space, the message format and the state machine specification are of relevance. In this section, we concern ourselves with the former and present an approach for constraining the input in accordance with a message format specification, thereby reducing the search space and ensuring that deeper parts of the tested code are reached earlier.

There is a large body of prior work on fuzzing of applications expecting the generated input to satisfy complex input format specifications [7, 202, 72, 141, 12]. Contrary to symbolic execution, fuzzing performs no formal reasoning and instead relies solely on randomly generated values to create test inputs, thus requiring even more time to satisfy input formats. Prior work on fuzzing attempts to address this problem by randomizing individual rules of a specified grammar [7, 202, 72] or individual fields of specified input blocks [141, 12]. Due to the lack of formal reasoning, it is necessary in both cases to manually provide a detailed description of the utilized input format, which can be cumbersome and error-prone. Inaccuracies in the provided input specification will cause the tested software to reject generated inputs early, i.e. without performing interesting input processing. We propose using symbolic execution—or more specifically—concolic testing (which combines fuzzing and symbolic execution) to ease the creation of input format specifications by allowing verification engineers to only partially specify the targeted structured input format. That is, unspecified parts of the input format can be treated as unconstrained symbolic values, thereby allowing an SMT solver to automatically fill in the leftover gaps based on extracted program constraints.

We present the Scheme-based Input Specification Language (SISL), an EDSL to programmatically specify structured binary input formats which are often used in security critical domains (such as the IoT). Furthermore, we illustrate that our proposed language can be easily integrated into existing concolic testing frameworks by presenting an exemplary integration for SymEx-VP (Chapter 3), a concolic testing engine for embedded RISC-V software. Lastly, we evaluate our EDSL by providing evidence that the minimal effort, required to create partial specifications, is outweighed by the gain in coverage and that our proposed EDSL is expressive enough to describe a wide range of structured binary input formats. To the best of our knowledge, SISL is the first input format specification language designed explicitly for symbolic execution. The SISL tooling is open source and can be obtained from `https://agra-uni-bremen.github.io/sisl/`.

6.1.1. Scheme-based Input Specification Language

SISL is an EDSL [92] for partially specifying parameterizable binary input formats for concolic software testing. As the name suggests, SISL is based on and embedded into the Scheme [168] programming language which, in turn, is a Lisp dialect. Conceptually, SISL is similar to the LibRISCV EDSL which is embedded into the Haskell programming language and was presented in Section 4.1. For SISL, we choose Scheme as the basis of our language since it supports hygienic macros which allow defining custom syntactic constructs within the language framework, thereby easing the creation of EDSLs [11].

Similar to block-based fuzzers [12, 141], SISL allows specifying binary input formats as a sequence of variable-width bit blocks. Contrary to existing work on fuzzing, SISL targets concolic testing and therefore supports distinct block types to distinguish concrete and symbolic values in the specified input format. Symbolic field values can optionally be constrained with symbolic expressions, hence allowing expressing the relationship between different symbolic fields (e.g. $X < Y$ must hold for two symbolic fields X and Y). Unconstrained symbolic fields can be used to leave parts of the input format unspecified, therefore allowing the concolic testing engine to fill in these gaps based on program execution and thus easing the creation of input format specifications. Defined input formats can also be nested, e.g. to express encapsulation in a network protocol context.

An example SISL input specification is provided in Listing 6.1 where a specification for the ICMPv6 message format is presented. ICMPv6 is a binary network protocol implemented on top of IPv6. For this reason, the SISL specification in Listing 6.1 defines two input formats. First, the IPv6 message format is defined in Line 1 - Line 9 using SISL's `define-input-format` keyword. This keyword defines a new input format and requires specifying the input format name, optional input format parameters, and the input format fields. In Line 1 the input format name is given as `ipv6-packet`, a `next-hdr` parameter is defined, and the special `&encapsulate` keyword is used to denote that the format encapsulates an additional `payload` format. In Line 2 - Line 9 the fields of the IPv6 packet format are defined. Each field definition takes at least two parameters: a field name (expressed as a Scheme symbol) and a field size in bits. Fields can either be concrete or symbolic. Concrete fields require the field value as a third argument. Symbolic fields support an optional third parameter to express symbolic constraints. For the `ipv6-packet` definition, six concrete fields are defined in Line 2 - Line 7. The majority of these fields (Line 3, Line 4, Line 6, and Line 7) use a concrete integer literal as a field value. The `version` field (Line 2) uses a predefined variable as a field value and the

```
1  (define-input-format (ipv6-packet next-hdr &encapsulate payload)
2    (make-uint 'version-field 4 ipv6-version-value)
3    (make-uint 'traffic-class 8 0)
4    (make-uint 'flow-label 20 0)
5    (make-uint 'payload-length 16 (input-format-bytesize payload))
6    (make-uint 'next-header 8 next-hdr)
7    (make-uint 'hop-limit 8 42))
8    (make-symbolic 'src-addr 128)
9    (make-symbolic 'dst-addr 128))
10
11 (define-input-format (icmpv6-packet &encapsulate body)
12   (make-symbolic 'type 8 `((Or
13                             (Eq type ,icmpv6-nbr-sol)
14                             (Eq type ,icmpv6-nbr-adv))))
15   (make-symbolic 'code 8)
16   (make-symbolic 'checksum 16))
17
18 (write-format
19   (ipv6-packet
20     icmpv6-next-header
21     (icmpv6-packet
22       (make-input-format
23         (make-symbolic 'body (bytes->bits 32))))))
```

Listing 6.1.: Excerpt of a SISL input specification for the ICMPv6 message format.

value of the `payload-length` field depends on the byte size of the `payload` parameter. Furthermore, the `ipv6-packet` definition also uses two symbolic fields for the source and destination address of the IPv6 header format (Line 8 - Line 9). IPv6 addresses have a complex internal structure which is cumbersome to express, by declaring them as symbolic the correct value for these fields will be inferred by the concolic testing engine during execution.

The second input format, defined in Listing 6.1, is the ICMPv6 message format (Line 11 - Line 16). This definition is analog to the `ipv6-packet` definition, with the exception that it only consists of symbolic fields (Line 12 - Line 16). Furthermore, the symbolic `type` field (Line 12 - Line 14) demonstrates the expression of symbolic constraints on a symbolic field values. Symbolic constraints are expressed as a list of KQuery

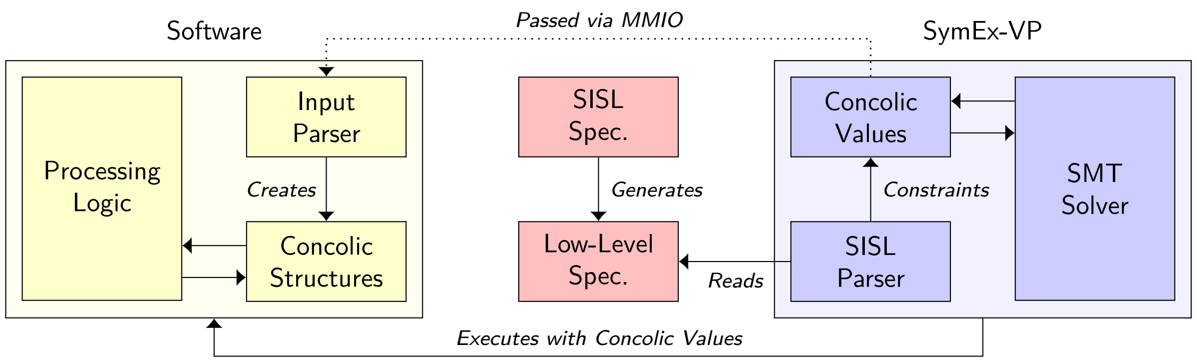


Figure 6.1.: Overview of our SISL-based concolic testing setup using SymEx-VP.

expressions, a textual representation of symbolic constraints from prior work [32]. In Line 12 - Line 14, the `type` field is constrained so that it either has the value of the variable `icmpv6-nbr-sol` or `icmpv6-nbr-adv`. These two variables refer to constants from the IPv6 NDP specification, thereby enabling targeted concolic testing of an NDP implementation with this SISL specification.

In order to enable such tests, the two described input formats are instantiated in Line 18 - Line 23 of Listing 6.1 with specific parameters as part of the `write-format` procedure invocation. In this case, the `next-hdr` of the `ipv6-packet` is instantiated with the value of the variable `icmpv6-next-header` and the `payload` parameter is set to an instance of an `icmpv6-packet` which itself has its `body` parameter set to an input format with 32 unconstrained symbolic bytes.

6.1.2. Overview and Implementation

We have integrated our proposed EDSL with SymEx-VP as presented in Chapter 3. An overview of the interaction between SISL, the tested software, and SymEx-VP is provided in Figure 6.1. The central component of Figure 6.1 is the high-level SISL specification written in Scheme. As discussed in Subsection 6.1.2, this specification is created manually by a verification engineer. Based on the human-readable SISL input specification, a machine-readable low-level specification is automatically generated. This low-level specification is then provided to and read by SymEx-VP, which constrains utilized concolic values according to the specification. Since SymEx-VP targets embedded RISC-V software in binary form, the constrained concolic input values are passed to the executed software via MMIO peripheral interfaces, e.g. via a network peripheral (see Section 3.1). The software binary is then explored by SymEx-VP based on these input

Table 6.1.: Comparison of concolic testing with SISL and the original SymEx-VP.

Application		SISL			SymEx-VP	
Name	ALOC	SLOC	#Paths	Solver Time	#Paths	Solver Time
Zephyr-CoAP	25383	24	23411	226 min	22999	232 min
Zephyr-IPv6-NDP	31066	30	15122	338 min	1736	453 min
Zephyr-MDNS	31238	35	19585	242 min	2287	452 min

values. The left side of Figure 6.1 shows a schematic representation of relevant software components performing input processing. Conceptually, the input parser of the software will process the concolic inputs and create data structures based on them. Since the inputs are concolic, the created data structures will also contain concolic values. Based on these concolic values, execution paths through both the input parser and the software processing logic (which processes data structures created by the input parser component) will be enumerated by SymEx-VP. Recall that SymEx-VP employs a standard DSE concolic testing technique where branches in the software are tracked and negated by an SMT solver to discover new assignments for concolic input values (see Section 2.3).

By constraining concolic input values prior to execution using SISL, we can (a) reduce the amount of generated input values which are rejected by the software's input parser early on and do not reach the processing logic and (b) reduce the amount of time spent in the SMT solver by using partially instead of fully symbolic inputs, thus reducing the complexity of SMT queries.

6.1.3. Evaluation

We evaluate our proposed input specification language by applying it to Zephyr[1]. Zephyr is a popular operating system for programming constrained embedded devices in the IoT. For this reason, Zephyr provides input handling routines for structured binary input formats used by different network protocols in this domain. We performed experiments with three different protocol message formats (CoAP, IPv6 NDP, mDNS) using example Zephyr applications. Generated input values were passed directly to the Zephyr network stack through a SLIP [155] network peripheral provided by SymEx-VP. The results

[1] `https://zephyrproject.org/`

of our experiments are shown in Table 6.1. For each application, we list the amount of RISC-V assembler instructions (ALOC) in the binary and the amount of SISL lines (SLOC), required for the created input format specification, as a complexity metric. We executed each application for 8 h using the created input specification with our SISL enhanced version of SymEx-VP and with the original SymEx-VP (i.e. entirely unconstrained symbolic input). For both executions, we list the amount of discovered paths through the program (as a coverage metric, column: #Paths) and the amount of time spent solving constraints on symbolic values (a known bottleneck of concolic testing, column: Solver Time).

The results in Table 6.1 demonstrate that partial SISL input specifications reduce the amount of solver time, thereby allowing the discovery of more execution paths through a given program in a given time span. The gain in path coverage increases with application complexity (as measured in assembler instructions, column: ALOC). We deem the effort required to create partial input specifications to be comparatively low, since complex parts of the input format can be marked as unconstrained symbolic and will thus be inferred during execution. For example, even for a complex input format like mDNS (which is encapsulated in an IPv6 and UDP packet) only 35 lines of SISL specification were required. The utilized SISL specifications and Zephyr applications are available as part of the artifacts [189].

6.1.4. Conclusion

In this section, we have presented SISL, an EDSL for creating input format specifications for testing network protocol implementations. SISL is based on the Scheme programming language and eases the creation of input format specifications by allowing the format to be partially specified. By combining these specifications with concolic testing, left-over gaps are treated as unconstrained symbolic values. Therefore, they are subject to symbolic reasoning based on extracted program constraints. This eases the creation of new protocol specifications. Conducted experiments with Zephyr indicate that our EDSL is expressive enough to support different binary input formats and the manual labor required to employ our EDSL is outweighed by the benefits in terms of increase in path coverage.

6.2. Symbolic Execution of Stateful Network Protocols

In Section 6.1, we have presented a specification language for describing inputs formats in order to optimize input generation, during symbolic execution, for testing network protocol implementations. Unfortunately, popular network protocols used in the IoT domain (e.g. MQTT-SN [176]) are stateful. For this reason, their expected message format differs depending on the current state of the protocol state machine (i.e. prior messages). As such, stateful network protocol implementations cannot be tested using the approach described in Section 6.1. In this section, we enhance this approach to support stateful network protocol implementations. Such implementations require multiple packets to be sent—within the same session—to be comprehensively tested. When sending only a single packet, deeper parts of the implementation are not reached as state (required to reach certain parts of the code) is not established and thus critical errors may be missed. Reasoning about multiple packets further increases the state space, thereby aggravating path explosion problems.

There is some prior work on symbolic execution of network protocol implementations [8, 34, 6, 174, 173]. Unfortunately, only few existing works concern themselves with testing of stateful network protocol implementations [6, 174, 173]. Therefore, the majority of prior work does not mitigate path explosion issues caused by the stateful nature of these network protocols. The closest related work is SymbexNet by Song et al. which proposes multi-packet exchange symbolic execution [173]. However, SymbexNet is not aware of the protocol state machine and while it can reason about multiple packets, it treats parts of the packet sequence (every packet except the last one) as concrete to mitigate state explosion issues. Heavy reliance on concretization can cause SymbexNet to miss paths and therefore bugs in the tested software.

In order to symbolically execute stateful network protocol implementations, we enhance our work presented in Section 6.1. For this purpose, we extend the EDSL to also describe the network protocol state machine, in addition to the message format. This enables us to explore stateful network protocol implementations which require multiple input packets to be sent in order to be comprehensively tested. Furthermore, we contribute a state-aware exploration algorithm for symbolic execution of stateful network protocols. This algorithm is based on protocol specifications created using our EDSL. We employ our approach to test different stateful network protocol implementations provided by two popular IoT operating systems (RIOT and Zephyr). Our experiments indicate that our approach achieves high code coverage with minimal specification and

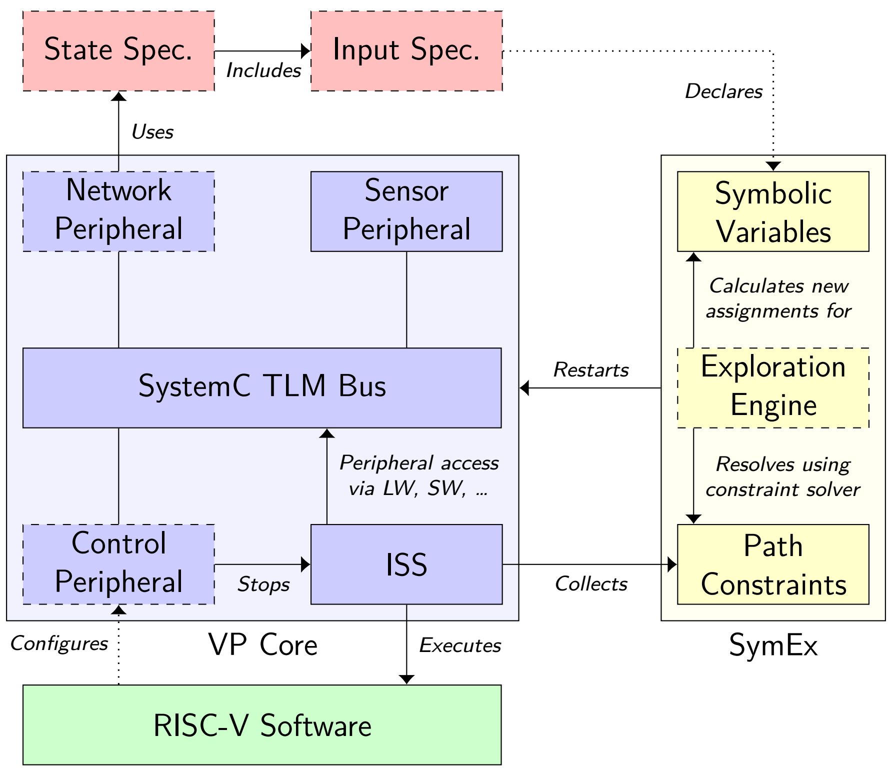


Figure 6.2.: Overview of our specification-based symbolic execution approach.

integration effort. The effectiveness of our approach is further affirmed by the fact that we managed to uncover three critical and previously unknown bugs in network protocol implementations of the RIOT operating system.

6.2.1. Approach

In order to overcome challenges regarding state space explosion, during the execution of stateful network protocols, we present a specification-based symbolic execution approach which is specifically tailored to embedded IoT applications.

6.2.1.1. Overview

An overview of our proposed approach is provided in Figure 6.2. Naturally, our implementation is based on SymEx-VP as presented in Chapter 3. The existing SymEx-VP architecture has been modified to implement support for specification-based symbolic execution of stateful network protocols. Components which have been modified for this purpose are shown in a dashed box in Figure 6.2. In total, we changed around 2500 LOC across 31 files in SymEx-VP.

In summary, the central components of the SymEx-VP architecture are the VP core and the symbolic execution abstraction (referred to as SymEx in Figure 6.2). The symbolic execution engine is responsible for test input generation based on tracked symbolic path constraints, the VP core is responsible for software execution and peripheral modeling. Both components have multiple subcomponents. The central subcomponent of the VP core is the ISS. The ISS is responsible for executing instructions of a software under test in 32-bit RISC-V machine code. The software, executed by the ISS, communicates with peripherals using load/store instructions (i.e. via MMIO). SymEx-VP models peripherals using SystemC TLM [180] which describes hardware peripherals based on transactions exchanged over a bus. For this reason, all modeled peripherals are attached to a TLM bus. In Figure 6.2, a network and a sensor peripheral are attached to the bus. Furthermore, a special control peripheral is available which allows the executed software to configure certain aspects of the performed symbolic execution via MMIO (see Subsection 6.2.1.3).

During software execution, the ISS collects constraints on symbolic values and communicates them to the symbolic execution engine. In order to implement the DSE algorithm introduced in Section 2.3, the executed software needs to signal the end of execution to the ISS. This is achieved via the symbolic control peripheral which then stops the ISS. After the ISS has been stopped, the exploration engine is used to determine new assignments for utilized symbolic variables. Lastly, software execution is restarted with these new values. This process is ideally repeated until all paths through the executed software have been explored.

We have enhanced this existing architecture to add support for specification-based symbolic execution of network protocols. For this purpose, we have modified a network peripheral provided by SymEx-VP to constrain inputs according to a provided input specification. Constrained inputs are partially symbolic, e.g. individual protocol fields may be declared as symbolic variables. The constrained input is passed through the TLM bus to the executed software via a SystemC TLM extension mechanism. As such, symbolic values originate in the network peripheral and no software modifications are required to inject them into the simulation. In order to facilitate testing of stateful network protocols, we combine the SISL input specification language (described in Section 6.1) with a novel state specification language. The latter allows constraining inputs differently based on the current protocol state. For example, an MQTT-SN implementation would expect different input packets depending on whether a connection was already established. More details on the specification languages are provided in Subsection 6.2.1.4 and

Subsection 6.2.1.5. Furthermore, the SymEx-VP exploration engine had to be modified to support symbolic execution with multiple input packets (Subsection 6.2.1.2). The algorithm used in this regard will be presented in the next section.

6.2.1.2. Multipacket Exploration

Stateful network protocol implementations establish program state based on received network packets. Therefore, it is vital to generate multiple partially symbolic network packets in order to comprehensively test these network protocol implementations. Otherwise, code that requires prior state to be established in order to be reachable will not be tested sufficiently and critical bugs may be missed during testing. Instead of testing software with a single symbolic network packet, we propose using symbolic *packet sequences*. A packet sequence is a fixed-length sequential arrangement of k network packets. Based on symbolic packet sequences, we propose *multipacket exploration* of stateful network protocol implementations where the packet sequence length (i.e. the number of packets) is increased incrementally. Symbolic execution is then only performed up to a certain depth as given by the current packet sequence length k. That is, software execution is suspended after the executed software processed network packet k. We refer to paths where execution was suspended as *partially explored paths*. By increasing the sequence length incrementally, we can reduce the initial state space and limit the complexity of generated SMT queries, hence cutting down solver time. A similar approach is also utilized by bounded model checking [43] and prior work on SymbexNet [173].

Figure 6.4 illustrates reachability of execution paths based on the current packet sequence length using a symbolic execution tree where each node represents a branch condition. For each branch condition, we track the path constraints (omitted in Figure 6.4) and the packet sequence length k with which it was initially discovered. For each packet sequence with length $k \geq 1 \land k \leq 3$ a subset of possible execution paths is shown in Figure 6.4. The red nodes in Figure 6.4 indicate the point in time at which the software has signaled that it has finished processing of a network packet and is about to process the nth network packet next. Software execution is suspended at this point if n exceeds the current search depth k. Execution paths for which execution was suspended are classified as partially explored and re-explored later when the search depth k is increased.

After suspending execution, we run the DSE algorithm from Section 2.3, thereby exploring execution paths reachable with the current packet sequence length k. This allows us to explore the execution tree in its breadth, we explore its depth by incrementally

```
$k \leftarrow 1$
$vars \leftarrow random\_values()$
while $true$ do
    while $\neg is\_empty(vars)$ do
        $execute(k, vars)$
        if $coverage\_stagnant()$ then
            $break$
        end if
        $vars \leftarrow negate\_random\_branch(k)$
    end while

    $k \leftarrow k + 1$
    while $partial\_path\_exists(k - 1)$ do
        $vars \leftarrow random\_partial\_path(k - 1)$
        $execute(k, vars)$
        if $coverage\_stagnant()$ then
            $break$
        end if
    end while

    $vars \leftarrow negate\_random\_branch(k)$
end while
```

Figure 6.3.: Multipacket exploration algorithm for symbolic software execution.

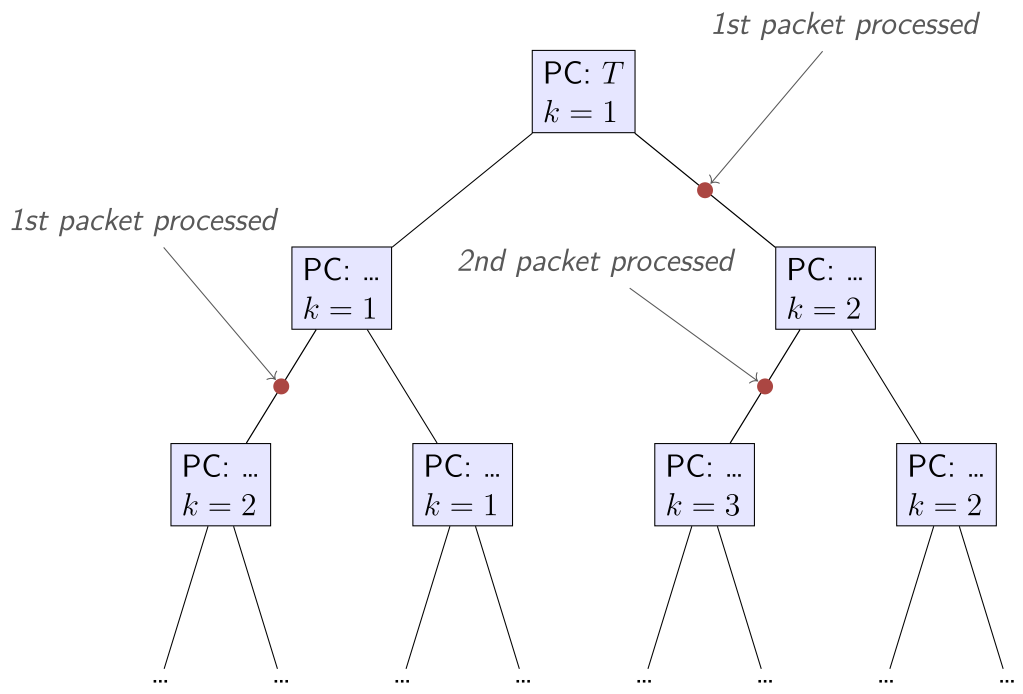


Figure 6.4.: Execution tree for exploration with packet sequence of length k.

increasing the packet sequence length k. The algorithm we propose in this regard is shown in Figure 6.3. In Line 2, we initialize all symbolic values with random values (as required by the DSE algorithm from Section 2.3). Afterward, we enter the exploration loop, which is executed until a timeout occurs (Line 3 - Line 22).

The exploration loop performs the following computations:

1. *Exploration with current sequence length.* The exploration starts out by executing the software with the configured variables $vars$ and stops when the kth packet has been processed by the software (Line 5). During execution, all branches which depend on symbolic variables are collected and for each branch the sequence length k, with which it was initially discovered, is tracked. After execution has been suspended, a random branch for a packet sequence of length k is selected, negated, and symbolic variables are assigned to trigger this branch (Line 9). For this purpose, we currently use a depth-first search algorithm, a survey of algorithms for selecting a branch to negate is provided by Baldoni et al. [10, Section 2.2]. This process is repeated until the coverage is stagnant (Line 6 - Line 8) or all branches for a packet sequence of length k have been negated, in which case $vars$ would be empty (Line 4).

2. *Re-execution of partial paths.* After exploring the execution tree in its breadth, the packet sequence length is incremented (Line 12) to explore its depth. In this regard, the algorithm re-executes paths which were only partially explored with $k = k - 1$. That is, it reuses the variable assignment which previously lead to suspension of software execution (Line 14) and restarts the execution until the next partial termination (Line 15). This allows initial discovery of branches which are only reached after processing an additional packet. The process is repeated until the coverage is stagnant (Line 16 - Line 18) or all partially explored paths have been re-executed (Line 13).

3. *Exploration with increased sequence length.* After discovering initial branches for a sequence of length k, the execution tree is explored in its breadth again by negating an initial random branch for $k + 1$ (Line 21) and then resuming execution at 1). Thereby exploring code in the protocol implementation which can only be reached after sending at least $k + 1$ network packets.

In summary, the algorithm explores paths reachable with a fixed search depth of k network packets and increments k whenever coverage stagnation is detected. Regarding the detection of coverage stagnation, we use a metric based on branch coverage where we consider exploration to be stuck when no new branches have been discovered within a configured amount of execution paths.

6.2.1.3. Control Peripheral

Since we execute bare metal embedded software directly, we cannot rely on common operating systems abstractions (such as system calls or sockets). Therefore, we need to instrument the executed RISC-V software to detect successful processing of symbolic network packets, as retrieved from the provided network interface. This is necessary as we need to determine when the executed software finished processing of an input packet in order to implement the algorithm from Subsection 6.2.1.2. For this purpose, we use the control peripheral, which is a small SystemC peripheral that allows the executed RISC-V software to communicate with the symbolic execution engine. By writing to a memory-mapped register, the executed software can indicate one of the following execution conditions:

1. *Packet processed.* A single network packet was processed and the software will now attempt to retrieve an additional network packet.

2. *Termination.* A network packet was processed but no further packet will be retrieved (e.g. because the connection was closed).

3. *Error.* An error state was reached, i.e. the software panic handler was invoked (e.g. due to a division by zero).

Executed paths are classified by the symbolic execution engine according to the indicated condition. Paths with condition 1) are considered partially explored and treated according to the algorithm described in Subsection 6.2.1.2, paths with condition 2) or 3) are not explored further. If a path with an error condition is encountered, this error is communicated to the verification engineer, who can then investigate it further. More information regarding software instrumentation is provided in Subsection 6.2.2.1.

6.2.1.4. Input Format Specification

As discussed in Subsection 6.2.1.2, our exploration algorithm is based on symbolic packet sequences. Unfortunately, it is infeasible to make all network packets of a symbolic packet sequence unconstrained symbolic since this would result in complex path constraints which will not be solvable in a timely manner. Instead, our approach relies on partially symbolic network packets. For this purpose, we leverage the SISL Scheme-based EDSL, which was presented in Section 6.1, and allows expressing network protocol formats based on variable-length bit fields. Using this EDSL, individual fields of a network protocol can be declared as symbolic by a verification engineer. This allows targeted testing of network protocol implementations and reduces the complexity of path constraints. In the aforementioned prior work, the EDSL has only been applied to test protocols with a single input packet. We enhance the EDSL with state specification facilities (presented in Subsection 6.2.1.5) to not only describe the protocol message format, but also the protocol state machine. Thereby enabling tests of stateful protocol implementations which require multiple packets to be sent in order to be comprehensively tested. In the following, we describe how the EDSL can be used to describe the message format of the MQTT-SN protocol as a prerequisite for the enhancements presented in Subsection 6.2.1.5.

An example input specification for the `SUBACK` message from the MQTT-SN [176] protocol is provided in Listing 6.2. In the IoT, MQTT-SN messages are commonly encapsulated in UDP [146] datagrams. Therefore, the example input specification from Listing 6.2 contains both a UDP (Line 1 - Line 5) as well as an MQTT-SN `SUBACK` (Line 7 - Line 15) input format specification. First, the UDP message format is defined

```
 1  (define-input-format (udp-fmt dst-port &encapsulate body)
 2    (make-uint 'src 16 2342)
 3    (make-uint 'dst 16 dst-port)
 4    (make-uint 'size 16 (+ 8 (input-format-bytesize body)))
 5    (make-symbolic 'cksum 16))
 6
 7  (define-input-format (suback-fmt msg-id)
 8    (make-uint 'len 8 8)
 9    (make-uint 'type 8 mqtt-suback)
10    (make-uint 'flags 8 0)
11    (make-symbolic 'topicid 16)
12    (make-uint 'msgid 16 msg-id)
13    (make-symbolic 'code 8 `((And
14                               (Uge ,code 0)
15                               (Ule ,code 3)))))
16
17  (write-format
18    (udp-datagram
19      1883 ;; MQTT-SN port
20      (suback-fmt
21        2342)))
```

Listing 6.2.: Excerpt of an input specification for MQTT-SN `SUBACK` messages.

in Line 1 - Line 5 using the `define-input-format` keyword. This keyword defines a new input format and requires specifying the input format name, optional input format parameters, and the input format fields. In Line 1, the input format name is given as `udp-fmt`, a `dst-port` format parameter is declared, and the `&encapsulate` keyword is used to denote that the UDP input format encapsulates an additional `body` format. Afterward, the UDP message format fields are declared in Line 2 - Line 5. Each field definition takes at least two parameters: a field name (expressed as a Scheme symbol) and a field size in bits. Fields can either be concrete or symbolic. Concrete fields require the field value as a third argument. Symbolic fields support an optional third parameter to express symbolic constraints. In accordance with the UDP protocol specification [146, p. 1], the specified UDP format consists of four 16-bit fields (Line 2 - Line 5). Of these four fields, only the last (`chksum`) is unconstrained symbolic. All other fields (`src`, `dst`, and `size`) are concrete. Regarding field values, `src` uses an integer literal, the `dst` value depends on the `dst-port` parameter, and the `size` field value is calculated from the `body` format size plus the size of the UDP format itself.

The format of the MQTT-SN `SUBACK` message is defined in Line 7 - Line 15. The definition is analog to the UDP message format definition described previously. However, the `SUBACK` format also demonstrates the definition of constrained symbolic fields. The `code` field declared in Line 13 - Line 15 is constrained to only represent symbolic values x with $x \geq 0 \land x \leq 3$. This matches the range of valid MQTT-SN return codes as specified in the MQTT-SN standard [176, Section 5.3.10]. Since both the UDP and the MQTT-SN `SUBACK` format depend on parameters, they need to be instantiated with concrete values for these parameters. For example, the `SUBACK` format depends on an `id` parameter which allows associating a client request with a broker response to this request. An example instantiation is provided in Line 18 - Line 21, where the UDP format is instantiated with a `dst-port` value of `1883` and encapsulates a `SUBACK` format. The `SUBACK` format itself is instantiated with the `msgid` value `2342`.

6.2.1.5. State Machine Specification

Since our proposed specification-based symbolic execution approach focuses on testing stateful network protocol implementations, it is insufficient to only consider input formats for a single message. As discussed previously, stateful network protocols implement a protocol state machine and expect different messages formats based on the current protocol state. In order to comprehensively test complex stateful network protocol implementations (like MQTT-SN), we enhance the aforementioned SISL input specification language with facilities for describing the protocol state machine. Conceptually, the enhanced version of the EDSL allows specifying the protocol state machine in R7RS Scheme [168]. This allows sending different partially symbolic network packets to the tested software based on the current protocol state. Transitions in the state machine are triggered based on network packets received from the tested software. For each state machine transition, a response message format is defined using the input specification language presented in Subsection 6.2.1.4.

An excerpt of the state machine specification for the MQTT-SN protocol is shown in Listing 6.3. Protocol state machines are defined using the `define-state-machine` keyword (Line 1). States for a state machine are defined using the `define-state` keyword (Line 4 - Line 21). The start state for the MQTT-SN machine (`pre-connected`) is defined explicitly in Line 2. Each defined state has a unique name and receives the network packet sent by the software in this state via the `input` parameter. Based on the

```
 1  (define-state-machine mqtt-machine
 2    (start pre-connected)
 3
 4    (define-state (pre-connected input)
 5      (switch (mqtt-msg-type input)
 6        ((CONNECT)
 7         (if (mqtt-will? input)
 8           (-> (make-resp will-topic-fmt) will-topic-req)
 9           (-> (make-resp connack-fmt) connected)))))
10
11    (define-state (will-topic-req input)
12      ...)
13
14    (define-state (connected input)
15      (switch (mqtt-msg-type input)
16        ((SUBSCRIBE)
17         (-> (make-resp (suback-fmt (msg-id input)))
18             subscribed))
19        ((DISCONNECT)
20         (-> (make-resp disconn-fmt)
21             disconnected))))
22    ...
23  )
```

Listing 6.3.: Excerpt of a state machine specification for the MQTT-SN protocol.

network packet input received by the software, different transitions can be triggered in each state. State machine transitions are defined using the `->` operator. For each state transition, the returned input specification format as well as the state the machine should transition to are specified. For example, in the `connected` state, the MQTT-SN message type is extracted from the input packet (Line 15). If the message received by the tested software is a MQTT-SN `SUBSCRIBE` message (Line 16), then the state machine responds with the `suback-fmt` input specification (as defined in Listing 6.2) and transitions to the `subscribed` state (Line 17 - Line 18). For this purpose, the `suback-fmt` specification is instantiated with the message identification extracted from the associated `SUBSCRIBE` message received by the client. Alternatively, it is also possible for the client to request a disconnect in the `connected` state. In this case, the state machine specification ac-

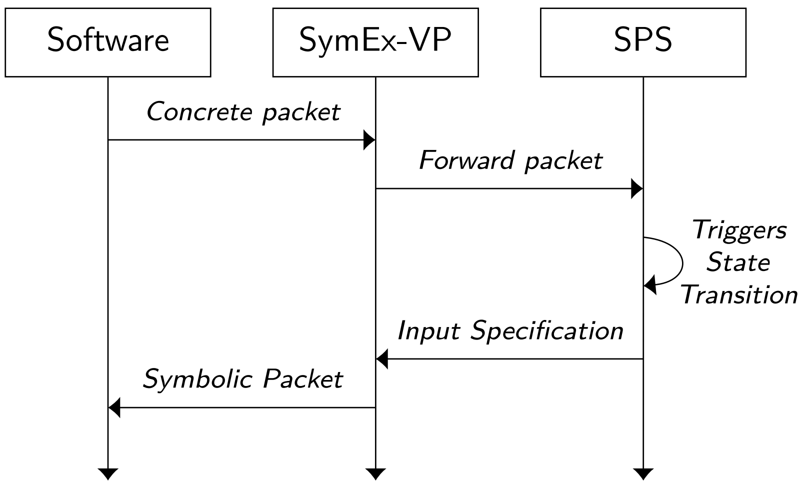


Figure 6.5.: Illustration of the communication with the State Protocol Server (SPS).

knowledges the disconnect with a `disconnect-fmt` input specification and transitions to the `disconnected` state (Line 20 - Line 21). Symbolic values are introduced into the simulation of the tested software in each state depending on the specification of the returned input formats using the EDSL described in Subsection 6.2.1.4.

In order to explore the tested software based on these symbolic values, we integrated the presented input and state specification languages with the network peripheral of SymEx-VP as illustrated in Figure 6.2. While the proposed state and input specification languages are based on R7RS Scheme, SymEx-VP itself is written in C++. In order to integrate these components, we have written a State Protocol Server (SPS) in Scheme which is responsible for triggering state machine transitions based on received input packets. SymEx-VP communicates with SPS using inter-process communication over a socket. The interactions between the tested software, SymEx-VP, and SPS are illustrated in Figure 6.5. The tested software initiates the communication by sending a concrete network packet via the network interface peripheral provided by SymEx-VP. SymEx-VP then intercepts this concrete packet and forwards it, in the form of a concrete byte vector, to SPS. SPS then executes the transitions for the current state based on the received input and returns the input specification (Subsection 6.2.1.4) in a low-level machine-readable format to SymEx-VP. SymEx-VP converts this low-level input specification format to a partially symbolic network packet and passes this network packet to the tested software via the network interface. The executed software is then explored based on the symbolic fields of the network packet according to multipacket exploration algorithm from Subsection 6.2.1.2.

6.2.1.6. Summary

In summary, we have presented a specification-based symbolic execution approach for stateful network protocol implementations. This approach is based on the SymEx-VP symbolic execution engine and is therefore specifically tailored to the IoT domain. By leveraging SystemC, we can model peripherals and sensors and thus comprehensively test IoT applications which communicate sensor data over stateful protocols like MQTT-SN. We contribute a new exploration strategy for SymEx-VP which tests software based on symbolic packet sequences up to an incrementally increased search depth (Subsection 6.2.1.2). In order to mitigate issues regarding state space explosion, we have utilized an EDSL which allows specifying symbolic fields within packets of a packet sequence (Subsection 6.2.1.4). Furthermore, we contributed a novel EDSL for describing protocol states machines, thereby varying the format of packets within a sequence based on input received by the tested software (Subsection 6.2.1.5).

6.2.2. Evaluation

We evaluate our specification-based symbolic execution approach by applying it to different network protocol implementations provided by two popular IoT operating systems. Our experiments show that we can achieve a significant increase in code coverage with minimal manual effort using our proposed approach. Additionally, we were also able to uncover previously unknown bugs in the tested implementations.

6.2.2.1. Experimental Setup

We perform experiments with stateful network protocol implementations provided by the aforementioned Zephyr and RIOT operating systems. For our experiments, we use our modified version of SymEx-VP (Subsection 6.2.1.1) in conjunction with a SLIP [155] network peripheral. SLIP is a protocol standardized by the IETF which allows transmitting Internet Protocol datagrams over a UART. Since SymEx-VP utilizes the SiFive HiFive1 hardware platform, we use a SystemC peripheral model of the SiFive UART for this purpose on the SymEx-VP side. Both RIOT and Zephyr already include a driver for the SiFive UART as well as an implementation of the SLIP protocol. As per Subsection 6.2.1.1, this allows us to inject symbolic values into the simulation of RIOT/Zephyr applications over the SystemC model of the SiFive UART using a TLM extension mechanism. The injected symbolic values represent partially symbolic network packets according to a provided specification (see Subsection 6.2.1.4 and Subsection 6.2.1.5). Tested applications are then explored based on a sequence of these packets according to the algorithm described in Subsection 6.2.1.2.

Since we inject symbolic values into the simulation via a SLIP network peripheral, we can keep modifications of tested software to a minimum. Purely for the purpose of injecting symbolic values, no software modifications are necessary. However, to employ our multipacket exploration algorithm (Subsection 6.2.1.2), the tested software needs to be instrumented in order to signal complete processing of an input packet. For this purpose, the tested software interacts via MMIO with the control peripheral described in Subsection 6.2.1.3. As such, we added a driver for the control peripheral to both RIOT and Zephyr and modified existing code to signal execution conditions via the driver to SymEx-VP. For example, we modified RIOT's panic handler to signal an error condition to SymEx-VP via the control peripheral. Since the MMIO interface of the control peripheral is structured around a single memory-mapped register, only around 100 LOC were added to RIOT and Zephyr for this purpose.

Regarding tested stateful network protocol implementations, we performed tests with MQTT-SN [176], DHCPv4 [62], and DHCPv6 [126]. Since all of these protocols require encapsulation, we have also created input specifications for the IPv6 [145], IPv6 [56], and UDP [146] protocols. In total, we have thus employed our proposed input specification languages for six different network protocols. Our experiments focus on the MQTT-SN protocol (for which we have tested two implementations) since we consider it the most complex and most popular stateful network protocol in the IoT domain. For each tested network protocol implementation, we started out by creating an input and state specification based on the protocol standard. Afterward, we modified SymEx-VP to measure code coverage based on executed instructions. We then executed tested protocol implementations using SymEx-VP and iteratively refined our input and state specification to maximize achieved coverage. This process took a graduate student about one working day per protocol.

In order to put achieved code coverage into perspective, we compare our coverage results with a baseline where the same applications are executed without employment of our state specification language. For our baseline, upper protocol layers are also specified but for the application layer protocol unconstrained symbolic data is returned and no different input formats are used based on the current protocol state. In both cases, the exploration algorithm described in Subsection 6.2.1.2 is used and coverage is considered stagnant if no new branch has been discovered within 50 execution paths. By using the same exploration strategy, memory model, et cetera, we can isolate the effects of the state specification language on coverage. This allows us to specifically assess the benefits of our state specification language, which we consider our most central contribution. For both setups, we executed our selected benchmark applications with a 2 h time budget on an Intel Xeon Gold 6240 running Alpine Linux Edge, results are presented in the next section.

6.2.2.2. Results

The results are shown in Table 6.2. In total, we have tested four benchmark applications (three RIOT and one Zephyr application). For RIOT, we tested two MQTT-SN implementations (`emcute` and `asymcute`) and RIOT's default DHCPv6 implementation. For Zephyr, we have performed experiments with the DHCPv6 implementation provided by the operating system. All information required to reproduce our results (including the utilized input and state specifications) is available as part of the artifacts [198].

The rows in Table 6.2 present the results for each benchmark application. As a complexity metric, we list the amount of RISC-V assembler instructions contained in the binary of each application (column: ALOC). As discussed in Subsection 6.2.2.1, we distinguish two configurations regarding symbolic execution: with and without our proposed state specification language. In each configuration, the benchmark applications were executed symbolically with a 2 h time budget. In order to compare these two configurations, Table 6.2 provides execution statistics for the symbolic analysis performed in this time span. Most importantly, achieved code coverage in the form of instruction coverage (column: *IC*), i.e. the percentage of executed unique RISC-V instructions.[2] Furthermore, Table 6.2 presents the amount of partially symbolic network packets sent in each configuration (column: *#Pkts*) and the amount of discovered execution paths (column: #Paths). For each benchmark, we then compare the difference between the two configurations regarding the aforementioned execution statistics. For the majority of tested applications (`emcute`, `asymcute`, and `dhcpv6`), use of our state specification language results in a roughly 20 % - 30 % increase in instruction coverage. An increase of 7 % can be observed for Zephyr's DHCPv6 implementation. Results are interpreted further in Subsection 6.2.2.3.

In order to illustrate how collected instruction coverage metrics develop over time, supplementary plots are provided in Figure 6.6. The provided plots are based on the

[2]Since both RIOT and Zephyr compile the entire operating system to a single binary we only measure coverage across source files relevant to the tested protocol implementation (as identified by text segment addresses).

Table 6.2.: Execution statistics for IoT protocol implementations with a 2 h time budget.

Application					
Benchmark	ALOC	Configuration	IC	#Pkts	#Paths
RIOT emcute	13983	w/o State Spec.	50 %	866	358
		w. State Spec.	83 %	2311	383
		Difference	+33 %	+1445	+25
RIOT asymcute	16703	w/o State Spec.	56 %	1418	562
		w. State Spec.	75 %	1752	434
		Difference	+19 %	−334	−128
RIOT DHCPv6	16203	w/o State Spec.	40 %	1170	419
		w. State Spec.	65 %	1340	470
		Difference	+25 %	+170	+51
Zephyr DHCPv4	18519	w/o State Spec.	57 %	1051	413
		w. State Spec.	64 %	1079	405
		Difference	+7 %	+28	−8

same data used for Table 6.2, however, the table only provides instruction coverage results after termination and does not illustrate how this coverage increases over time. In Figure 6.6, one plot is provided for each benchmark application. For each plot, the accumulated instruction coverage is specified on the y-axis while the current execution path is given on the x-axis. As such, each plot shows the accumulated instruction coverage achieved after execution of each listed execution path. Within each plot, we compare the aforementioned configurations. Instruction coverage without a state specification is plotted in blue while coverage with employment of the state specification language is plotted in red. For all benchmarks, coverage diverges towards 100 % with periods of stagnation in between.

6.2.2.3. Interpretation

Our results show that a significant increase in code coverage can be achieved through employment of specification-based symbolic execution using our proposed state specification language. For implementations of the popular MQTT-SN IoT protocol, we were able to achieve a 33 % and 19 % increase in code coverage for two RIOT implementations (`emcute` and `asymcute`). Hence, demonstrating that the increase in code coverage is not specific to a singular implementation of the MQTT-SN protocol. We deem the increase to be significant and to outweigh the manual effort required to create the required input

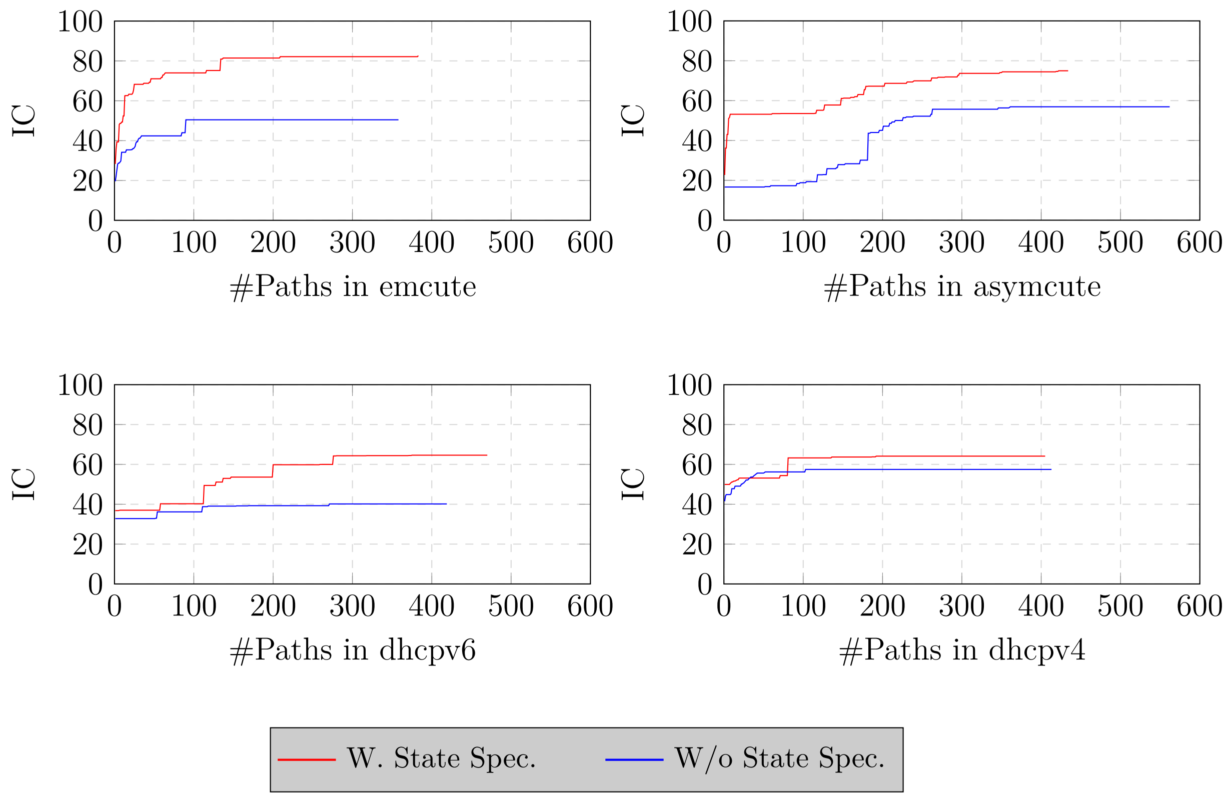


Figure 6.6.: Plots comparing achieved instruction coverage over time for symbolic execution with and without employment of our state specification language.

and state specifications, especially considering that—once created for a given protocol—they can be reused for different implementations of that protocol. Furthermore, our experiments illustrate that our proposed specification languages are applicable to other protocols by performing tests with a DHCPv4 and a DHCPv6 implementation. For DHCPv6, we achieved a similar increase in code coverage of 25 %. For DHCPv4, the increase in coverage is only 7 % this is because the DHCPv4 protocol has fewer messages types and is thus less complex than DHCPv6 or MQTT-SN. The data in Table 6.2 also illustrates that sending more packets or executing more paths does not necessarily result in higher coverage. For example, for the emcute benchmark, an increase in instruction coverage by 19 % can be observed, although 334 fewer packets were sent, and 128 fewer paths were discovered. This serves to show that our proposed approach results in the discovery of more “interesting” paths, i.e. paths which result in higher instruction coverage. As plotted in Figure 6.6, employment of our state specification language results in an increased code coverage at any point in time during symbolic execution. An exception in this regard is only the tested DHCPv4 implementation, which briefly achieves a minimally higher coverage without our state specification. We attribute this divergence to the fact that the provided state specification might be too narrow given the smaller state space of the protocol. In terms of re-usability, we were also able to apply our specification-based symbolic execution approach to different operating systems in the IoT domain (RIOT and Zephyr).

6.2.2.4. Encountered Errors

As discussed in Chapter 5, detection of common programming errors (such as memory corruptions) is difficult on embedded systems as they do not support runtime protection mechanisms commonly available on conventional devices. For this reason, faults caused by programming errors are often not detected at runtime [128]. Unfortunately, we have not yet integrated the error detection techniques, presented in Chapter 5, with our specification-based symbolic execution approach. This is because, in this section, we focus primarily on code coverage for the symbolic execution of stateful network protocols. Nonetheless, we were still able to uncover previously unknown bugs in RIOT as part of our conducted experiments (without any additional error detection techniques). All encountered bugs have been confirmed and fixed by RIOT developers. We further describe each encountered issue below and provide a reference to the bug report in the public bug tracker of the RIOT operating system:

- An out-of-bounds memory read occurring during option parsing in RIOT's DHCPv6 implementation. This issue was discovered as a load access fault was ultimately triggered, which caused an invocation of RIOT's panic handler.[3]
- On certain inputs, RIOT's asymcute module would not unlock a mutex on a return path. This causes a deadlock where asymcute is no longer able to process any further packets, thus allowing for a denial of service. The issue was discovered via a timeout mechanism in SymEx-VP.[4]
- When checking for acknowledgments of MQTT-SN packets, RIOT's asymcute implementation would not take the request type into account. Matching only occurred based on the message ID. As such, RIOT would, for example, match a `SUBACK` response to a `PUBLISH` request (which is incorrect). This caused a `NULL` pointer dereference in RIOT which resulted in a load fault and thus a panic handler invocation.[5]

In summary, our proposed methodology allows for a significant increase in code coverage with minimal manual effort for the symbolic execution of complex stateful network protocol implementations, like MQTT-SN, in the IoT. The fact that we were able to uncover three previously unknown bugs in the popular and well-tested RIOT operating system further affirms the effectiveness of our proposed symbolic execution approach.

6.2.3. Related Work

Dynamic automated testing of network protocol implementations is an active research area with a vast body of existing research. The majority of existing work utilizes fuzzing to automatically generate test inputs for network protocol implementations. Fuzzing-based approaches rely on random input generation and are thus often incapable of satisfying predefined structured input formats (such as network packet formats). In order to mitigate this issue, prior work on fuzzing mutates existing network packets, e.g. by transforming individual protocol fields [139, 80, 38]. Since this approach requires sniffing existing network packets, related work has also proposed fuzzing of network protocol implementations based on created protocol and input specifications [12, 107, 134]. Furthermore, prior work on fuzzing has experimented with inferring protocol state machines using machine learning techniques, thereby reducing manual effort [170, 182, 69].

[3]`https://github.com/RIOT-OS/RIOT/pull/18307`
[4]`https://github.com/RIOT-OS/RIOT/pull/18289`
[5]`https://github.com/RIOT-OS/RIOT/pull/18434`

Nonetheless, fuzzing remains limited by the inability to satisfy complex input constraints. Contrary to fuzzing, symbolic execution does perform formal reasoning on symbolic expressions and is thus also capable of solving more complex input constraints. Popular symbolic execution engines include KLEE [32], S^2E [41], or angr [169]. However, similar to fuzzing, the majority of existing work focuses on symbolic execution of software for conventional desktop systems and is thus not directly applicable to the embedded IoT domain. KLEE, for example, executes LLVM IR (the intermediate language of the LLVM compiler infrastructure) symbolically and thus does not allow execution of inline assembly, which is common in the embedded domain. Furthermore, KLEE focuses on environmental modeling for POSIX [95] systems and does therefore not support low-level interactions with hardware peripherals.

As discussed in Section 6.2, employment of symbolic execution for testing stateful network protocol implementations is limited by state space explosion issues. Existing work on SymNV addresses this problem by allowing developers to mark individual fields as symbolic [174]. SymNV is extended further in prior work to support exploration of stateful network protocol by combining concrete and symbolic execution and only treating the last packet of a packet sequence as symbolic [173]. Our proposed multipacket exploration algorithm retains symbolic values across the sequence and instead relies more heavily on input specification to reduce the state space. Furthermore, the aforementioned existing work is specific to conventional devices.

Prior work attempts to leverage existing symbolic execution engines for the embedded domain through a process called “rehosting” where peripheral behavior is only approximated up to a point where it is sufficiently accurate to execute a single firmware image [54, 45, 68]. That is, instead of modeling the entire hardware platform (as done in virtual prototyping with SystemC). A survey of existing rehosting techniques is provided by Fasano et al. [66]. To the best of our knowledge, there is no prior work which provides symbolic execution of stateful network implementations for embedded devices, either through rehosting (as proposed in prior work) or peripheral modeling (as performed in this section). The closest related work we are aware of is KleeNet [158] which provides symbolic execution for network protocols of the Contiki [64] IoT operating system. However, KleeNet only supports Contiki’s native target (i.e. x86) and is incapable of executing low-level code interacting with peripherals. Furthermore, since KleeNet executes tests on an architecture that is different from the production environment, it will miss architecture-specific bugs (e.g. misaligned loads).

6.2.4. Discussion and Future Work

In this section, we focused on increasing code coverage for stateful network protocol implementations through specification-based symbolic execution. Nonetheless, we were able to uncover three previously unknown bugs in the popular and well-tested RIOT operating systems. However, since prior work has shown that programming errors often remain undetected at runtime—without instrumentation—in the embedded IoT domain [128], it is advantageous to integrate our specification-based symbolic execution technique with error detection techniques. We believe this to be a complementary issue as, conceptually, error detection techniques check each path—explored through symbolic execution—for an error case. In this context, our specification-based symbolic execution approach is complementary as it focuses on increasing coverage on enumerated paths. In Section 5.3, we have presented an error detection technique for spatial memory safety violations, but have not yet integrated it with specification-based symbolic execution. Apart from spatial violations, it would also be interesting to detect functional errors in protocol implementations, which prior work has achieved through rule-specifications that could be integrated into our state specification language in future work [173, 174].

Regarding code coverage, our approach relies heavily on the input and state specifications created by verification engineers. Creating a good input and state specification is challenging. If the specification is too narrow, important paths may be missed and coverage may become stagnant quickly. If too much data is declared as unconstrained symbolic, then the complexity of path constraints increases and execution slows down since more time is spent in the SMT solver. For this reason, it is important to iteratively refine created input and state specifications. In order to ease this process, it would be worthwhile to provide a visualization of the performed symbolic execution which illustrates how fields of an input packet affect coverage, thereby assisting in the creation of protocol specifications. Initial work towards such a visualization is presented later on in Section 6.3.

A further limitation of our specification-based symbolic execution approach is the manual effort required to create input and state specifications. However, our experiments indicate that specifications can be partially reused. For example, we were able to reuse our UDP and IPv6 specifications for all experiments conducted with RIOT. Since our EDSLs are based on R7RS Scheme, we envision that re-usable specifications for common network protocols (like IPv6 or UDP) will be provided as R7RS libraries [168, Section 5.6].

Lastly, the experimental setup described in Subsection 6.2.2.1 could be improved further. Presently, we inject symbolic network packets via a SLIP-based network peripheral into the simulation. We used SLIP for this purpose as it is widely supported by existing IoT operating systems and trivial to implement. However, a drawback of SLIP is that it relies on byte stuffing, i.e. requires escaping of certain control bytes in the packet data. Unfortunately, expressing these escaping rules as symbolic expressions is cumbersome and increases their complexity. For this reason, we presently constrain input bytes to never match the four control bytes and thus cannot generate inputs containing them. This could be addressed in future work by using a network peripheral which does not rely on byte stuffing, e.g. Ethernet. Furthermore, it would be desirable to remove the need for firmware modifications from our experimental setup in future work, thereby enabling developers to perform tests on the exact same binary that will be used in production.

6.2.5. Conclusion

We have presented a novel specification-based symbolic execution approach for testing stateful network protocol implementations. Our proposed approach is specifically tailored to the low-end IoT domain. In this domain, we have conducted experiments with stateful network protocol implementations provided by the IoT operating systems RIOT and Zephyr. For complex stateful network protocols (like MQTT-SN) our experiments show that a significant increase in code coverage of 20 % - 30 % can be achieved using our proposed approach. The effectiveness of our approach is affirmed further by three critical and previously unknown bugs that we were able to uncover in network protocol implementations provided by the RIOT operating system. The central contributions of this work include: a novel exploration algorithm for symbolic execution of stateful network protocol implementations (Subsection 6.2.1.2) and an EDSL for specifying the state machine of stateful network protocols (Subsection 6.2.1.5). To stimulate further research on this topic, we have released the implementation of our proposed specification-based symbolic execution approach—and all associated tooling—as open source software.[6,7]

[6] `https://github.com/agra-uni-bremen/sps`
[7] `https://github.com/agra-uni-bremen/sps-vp`

6.3. Visualizing Symbolic Execution Results

Section 6.1 and Section 6.2 presented a specification-based symbolic execution approach which uses protocol specifications to optimize input generation for embedded IoT applications. A challenge in this regard is the creation of "good" protocol specifications. On the one hand, if the specification is too narrow (i.e. too strongly constrained), then bugs in the tested software will be missed because inputs triggering them will not be generated. On the other hand, if the specification is too broad (i.e. too loosely constrained), then bugs in the tested software will be missed because deeper parts of the implementation—where they most commonly occur—are not reached within a fixed time budget. For the experiments conducted in Subsection 6.2.2.1, we have incrementally refined created protocol specifications. In this section, we want to improve upon this proposed approach.

In order to refine a protocol specification, the symbolic analysis conducted with a prior version of the specification must be evaluated. Coverage metrics are commonly used to assess the quality of performed tests [108]. Therefore, we have also used them for this purpose in Subsection 6.2.2.1. However, while coverage metrics indicate which parts of the code base have not been tested sufficiently, they do not include information on why these parts have not been reached. A potential reason for incompleteness in the symbolic execution domain are concretizations, i.e. the conversion of a symbolic value to a concrete one. Concretizations provide a trade-off between completeness and efficiency and can, for example, be used when SMT queries become too complex for the solver to handle in a timely manner [53]. They impact completeness as branch conditions, executed after the concretization point, are only concretely evaluated and not subject to symbolic reasoning [10, Section 2.1]. In the context of the creation of protocol specifications, it is useful to identify where concretizations took place in order to refine the specification accordingly. For example, reducing the use of symbolic fields within a protocol specification and employing divide-and-conquer to avoid concretization due to query complexity. Prior work attempts to communicate properties of a performed symbolic analysis through graph-based visualizations [3, 81, 89]. Unfortunately, the aforementioned prior work has largely been evaluated using smaller example code and is not deemed suitable for larger applications, e.g. the network stack of the RIOT operating system.

We attempt to address scalability issues by instead relying on coverage metrics and attempting to augment them with additional symbolic execution specific information. While we believe this technique to be applicable to different properties of a symbolic execution analysis, we focus on concretization in this section. More specifically, we contribute a new coverage metric by enhancing the existing line coverage metric with information about concretizations. We refer to this combination as *concolic line coverage* in the following. Concolic line coverage identifies code parts where concretization initially occurs. Since concretization affects all lines which thereafter operate on the concretized value, we perform taint tracking to quantify all affected lines. Similar to existing front ends for coverage metrics, we also contribute a code-based visualization of concolic line coverage that eases identifying source lines affected by concretization. Based on the visualization, protocol specifications can be systematically refined to avoid concretization.

6.3.1. Concolic Line Coverage

Standard line coverage measures the amount of executed lines in percent. Concolic line coverage augments line coverage with concretization information. As explained in Section 6.3, concretization does not only affect a single line. Therefore, in order to identify all operations that depend on a concretized value, we employ *taint tracking*. A concretization of a previously symbolic value causes this value to become tainted. All operations, conducted on a tainted value, propagate this tainting information. Meaning, an operation receiving a tainted value as an input will also return a tainted value as an output. We refer to these operations as *tainted operations* in the following. Tainted operations allow identifying lines which depend—directly or indirectly—on a concretized value.

Based on tainted operations, concolic line coverage checks—for each executed line—if it has been executed at least once with a symbolic value (S), a concretized value (C), and/or a normal concrete value (N). Since each line is executed multiple times under potentially different path constraints, it is possible for the same line to satisfy multiple of the aforementioned properties.[8] Let LC be the set of executed lines, as measured by line coverage, then the additional information gathered by concolic line coverage is best formalized as the binary relation $R \subseteq LC \times \{S, C, N\}$. Using this relation, concolic line coverage itself can be defined over the set L of total source code lines with $LC \subseteq L$ as follows:

$$\frac{|\{l \in LC \mid (l, C) \notin R\}|}{|L|} \cdot 100$$

In summary, we distinguish lines executed with symbolic, concrete, and concretized values. In this context, concolic line coverage is the percentage of executed lines which

[8]Similarly, since path conditions differ in each execution, it is sufficient for a line to be executed once with a concretized value in order for the execution engine to potentially miss a path which is only reachable under the current path constraints.

have *not* been executed with a concretized value (as defined in Section 6.3). Full concolic line coverage is reached if all lines have been executed with either concrete or symbolic values (i.e. no line has been executed with a concretized value), in both cases it is guaranteed that no paths have been missed due to concretization.

6.3.2. Implementation

We have implemented our proposed coverage metric on top of the SymEx-VP symbolic execution engine (see Chapter 3). As described previously, SymEx-VP performs concolic execution of 32-bit RISC-V binary code. For an integration with concolic line coverage, we modified roughly 1000 LOC in SymEx-VP. We further describe performed changes below.

6.3.2.1. Taint Tracking

In order to implement the generation of concolic line coverage metrics, we had to add taint tracking support for concretization to SymEx-VP. This was achieved by modifying the data type used by the concolic execution engine to represent concolic values. Apart from a symbolic expression and a concrete value, our modified version of this data type also tracks whether the value is tainted. Accordingly, we also had to modify functions implementing operations on concolic values (e.g. addition or subtraction) as these operations need to propagate tainting information correctly. For example, an addition operation that adds a non-tainted and a tainted concolic value must itself return a tainted value as a result. Fortunately, SymEx-VP generates the majority of functions performing operations on concolic values from function templates; thus, we were able to keep modification to a minimum.

Implementing tainting for concolic values in SymEx-VP allows marking them as tainted when a concretization is performed. In the execution unit of SymEx-VP, it is then possible to identify tainted operations by iterating over all operands of an executed binary code instruction (as defined by the instruction type) and checking whether any of these operands are tainted.

6.3.2.2. Coverage Support

Apart from taint tracking for concretization, we also had to track coverage information in SymEx-VP. As per Subsection 6.3.1, concolic line coverage is based on standard line coverage. In order to track line coverage, we had to extract information about functions, source files, and source lines during execution of RISC-V machine code in SymEx-VP. Similar to debuggers such as GDB, we rely on the DWARF debugging format to extract information about the source code for executed machine code [55]. For this purpose, we added a coverage component to SymEx-VP. This component is passed the address of an executed instruction, including tainting information, and then extracts the aforementioned source code information using `libdwfl` from `elfutils`[9]. Based on the extracted information, the coverage component populates internal tables to track how often a given line in a given file/function has been executed and whether it includes a tainted operation.

After program execution terminates, tracked coverage information is stored in the JSON format on a per source file basis. More specifically, we utilize the JSON format used by GNU `gcov` and extended this format to store information about concretization and symbolic execution as discussed in Subsection 6.3.1.

6.3.2.3. Visualization

In order to visualize the collected concolic line coverage metrics, we implemented a custom front end for the aforementioned JSON format. This visualization front end receives the JSON files as an input and creates HTML files as an output. For each source file, an HTML page is generated which displays the source code and colors source lines according to their properties (concretized, executed, unexecuted). In this regard, it is intentionally similar to existing visualization for line coverage metrics. This should ease interpretation of concolic line coverage metrics as verification engineers are already familiar with such visual representations from the concrete testing context. The visualization is further described using an exemplary case study in Subsection 6.3.3, a screenshot is provided in Figure 6.7.

6.3.3. Case Study

This section evaluates our proposed approach by applying it to the RIOT operating system and thereby illustrates how concolic line coverage and the associated visualization can aid verification engineers in understanding the effects of concretization on their performed tests.

[9]`https://sourceware.org/elfutils/`

6.3.3.1. Concolic Execution Setup

In our case study, we test the implementation of a network protocol through concolic execution using our modified version of SymEx-VP. Similar to prior experiments conducted in this chapter, we utilize the RIOT operating system for evaluation purposes. More specifically, we make use of RIOT's CoAP implementation (called `nanocoap`). CoAP is a network protocol commonly used in this domain to implement client-server architectures [167].

For our performed tests, we wrote a test harness for `nanocoap` which passes a fixed-size buffer with concolic values to the parsing function of `nanocoap` (i.e. `coap_parse`). This test harness was executed with a short 5 min time budget. Regarding the concretization strategy, we utilized address concretization for our performed tests. Address concretization is a popular memory model for concolic execution, in this memory model, concretization is performed when a symbolic value is used to address memory. This reduces the complexity of tracked constraints but—like any form of concretization—can cause the engine to miss paths [10, Section 3.2]. In this exemplary case study, it is therefore important for the verification engineer to identify code parts where address concretization is performed, as well as identifying code parts that depend on a memory value loaded from a concretized memory address.

6.3.3.2. Test Results

Executing the described test harness with our modified version of SymEx-VP resulted in the discovery of 358 paths in our specified time budget. The generated concolic line coverage reported concretization in two files:

1. `nanocoap.c` which provides the `coap_parse` function targeted by our test harness. In this file 67 lines were executed and 32 of these executed lines depended on concretized values.

2. `byteorder.h` which provides utility functions for byte order conversions used by `nanocoap.c`. In this file 10 lines were executed and all of them depended on concretized values.

We then generated a visualization from the JSON files emitted by SymEx-VP. While inspecting this visualization, we focused on the `coap_setup` function from `nanocoap.c` as this is the entry point of our test harness. The relevant part of the visualization for this function is shown in Figure 6.7, it will be further explained in the following.

100	SC	**198**	`while (pkt_pos < pkt_end) {`
101	SC	**1119**	`    uint8_t *option_start = pkt_pos;`
102	SC	**1119**	**`    uint8_t option_byte = *pkt_pos++;`**
103	SC	**1119**	`    if (option_byte == 0xff) {`
104	SC	**13**	`        pkt->payload = pkt_pos;`
105	SC	**13**	`        pkt->payload_len = buf + len - pkt_pos;`
107		**13**	`        break;`
108			`    }`
109			`    else {`
110	SC	**1106**	`        int option_delta = _decode_value(option_byte >> 4, &pkt_pos, pkt_end);`
111	SC	**1106**	`        if (option_delta < 0)`
113		**12**	`            return -EBADMSG;`
115	SC	**1094**	`        int option_len = _decode_value(option_byte & 0xf, &pkt_pos, pkt_end);`
116	SC	**1094**	`        if (option_len < 0)`
118		**13**	`            return -EBADMSG;`
120	SC	**1081**	`        option_nr += option_delta;`
122			
123	SC	**1081**	`        if (option_delta) {`
124		**189**	`            if (option_count >= CONFIG_NANOCOAP_NOPTS_MAX)`
126		✗	`                return -ENOMEM;`
128			
129	SC	**189**	`            optpos->opt_num = option_nr;`
130	SC	**189**	`            optpos->offset = (uintptr_t)option_start - (uintptr_t)hdr;`
132		**189**	`            optpos++;`
133		**189**	`            option_count++;`
134			`        }`
135			
136	SC	**1081**	`        pkt_pos += option_len;`
137			`    }`
138			`}`

Figure 6.7.: Concretization visualization for a simplified option parsing loop in RIOT's `nanocoap` CoAP implementation.

6.3.3.3. Visualization

The visualization in Figure 6.7 consists of four columns: the source line number, a source line classifier, the amount of times the line has been executed, and the source code.[10] As per Subsection 6.3.1, the classifier specifies if a line has been executed at least once with a symbolic value (**S**) and/or a concretized value (**C**). Additionally, the source code lines are colored. Green lines have been executed at least once, red lines have not been executed at all, and blue lines have been executed at least once with a concretized value. Source lines that create a new concretized value from a previously symbolic value (i.e. perform address concretization) are marked with a bold font.

The code in Figure 6.7 is responsible for CoAP option parsing. A CoAP option consists of an option number and an option value. The length of the option value is specified by an option length. The option number and an option value are both encoded using

[10]The code has been reformatted for clarity, thus there are line number gaps. Please also note that the discrepancy in the execution count between loop header (Line 100) and body (Line 101) are due to the fact that the loop header counts how often the loop is entered and not loop iterations.

a single byte [167, Section 3.1]. This byte is extracted in Line 102 by dereferencing a pointer into the buffer of concolic values as created by the test harness. Afterward, the individual components (option number and option value) are extracted in Line 110 - Line 134. Finally, the pointer is advanced in Line 136 to parse the next option.

The visualization indicates that concretization occurs initially in Line 102, all other blue-colored lines are tainted because they depend indirectly on a value concretized in Line 102. Concretization in Line 102 must occur due to the utilized address concretization memory model. For this to be the case, the `pkt_pos` variable in Line 102 must be a symbolic value. This occurs when the previous loop iteration adds `option_len` to `pkt_pos` in Line 136 because `option_len` is a symbolic value read from the buffer allocated by the test harness. In this case, `pkt_pos` would be symbolic in the next loop iteration and dereferencing it in Line 102 will result in concretization due to the utilized memory model. All other tainted lines depend, directly or indirectly, on the concretized `pkt_pos` value.

This case study illustrates how concolic line coverage and our generated visualization can aid a verification engineer in quickly identifying sources of concretization and their impact on concolic execution. In the illustrated scenario, the verification engineer could then change the configuration of the concolic execution engine to use a different memory model to reduce concretization. A survey of different memory models is provided by Baldoni et al. [10, Section 3].

6.3.4. Related Work

Prior work utilizes interactive graph-based visualizations to communicate properties of the performed symbolic execution to the verification engineer. Hentschel et al. present a Symbolic Execution Debugger (SED) which enables interactive debugging of Java programs and visualizes different paths through the program using execution trees [81]. Additional related work presents other graph-based visualizations without focusing on interactive debugging. In this regard, Honfi et al. present SEViz [89] which generates symbolic execution trees for .NET programs and Angelini et al. present SymNav [3] which generates an interactive visualization consisting of different interface elements including control flow graphs. Compared to these prior approaches, our own work differs in two core aspects: (1) we focus on concolic execution instead of symbolic execution thereby visualizing different aspects of the performed analysis (namely concretization instead of propagation of symbolic values and discovered paths) (2) instead of utilizing graph-based visualizations we propose a code-based visualization using custom coverage metrics, thereby easing application to larger codebases by quantifying performed concretizations.

6.3.5. Conclusion

In conclusion, we have presented a coverage metric and a visualization of this metric, which eases evaluating the results of a performed symbolic analysis. As such, these techniques can guide verification engineers in the process of refining manually created protocol specifications for the specification-based symbolic execution approach presented in Section 6.1 and Section 6.2. The case study from Subsection 6.3.3 demonstrates the usefulness of our proposed metric and the associated visualization. We plan to further improve this visualization in future work by augmenting it with additional information. In this regard, one envisioned enhancement would be identifying concretized variables in an executed source line. Presently, the entire source line is highlighted and it is up to the verification engineer to identify which variables in that highlighted line are subject to concretization. Our experiments with the RIOT operating system indicate that—relative to the amount of executed lines—only a minority of lines are executed using concretized values when employing address concretization. In this regard, our visualization helps to identify these lines and thereby eases identifying the effects of concretization on tested code. In order to allow others to make use of our approach, we have released both our modified version of SymEx-VP and our visualization front end as open source on GitHub.[11,12]

[11]`https://github.com/agra-uni-bremen/coverage-vp`
[12]`https://github.com/agra-uni-bremen/jcovr`

Chapter 7.

Conclusion

Symbolic execution is an automated software testing technique that has yielded promising results for finding software bugs. Unfortunately, its application to embedded firmware is presently limited due to unique challenges associated with this domain [207, 128, 204]. This thesis presented a novel approach for addressing these challenges. To this end, we contributed an integration with SystemC hardware models, formal ISA semantics, custom error detection techniques, and input generation heuristics for testing IoT network protocol implementations. Compared to prior work, these contributions enable an accurate analysis that is faithful to both the ISA specification (through formal semantics) and the peripheral behavior (through SystemC hardware models). In the following, contributions will be summarized and opportunities for future work will be laid out.

7.1. Summary

Chapter 3 concerned itself with challenges related to environment modeling. Software does not operate in a vacuum; it interacts heavily with its surrounding environment. In the embedded domain, these interactions are performed through hardware peripherals. Since symbolic execution is a dynamic testing technique, it needs to support these interactions. For this purpose, Chapter 3 contributed an integration of symbolic execution with SystemC TLM hardware models by facilitating VPs. This proposed approach is implemented in SymEx-VP, a VP performing binary-level symbolic execution of 32-bit RISC-V [153, 154] machine code. SymEx-VP is binary-compatible with the SiFive HiFive1 hardware platform and performed experiments demonstrate that it can symbolically analyze different firmware images targeting this platform. Symbolic values can be injected into firmware simulation through the hardware peripheral interface, using a presented SystemC TLM extension, thereby avoiding firmware modifications for input injection purposes. Existing SystemC models can be easily integrated with this TLM extension using a proposed overlay mechanism.

Apart from hardware models, SymEx-VP also provides a symbolic ISS to execute RISC-V binary code instructions with symbolic operands. This ISS was manually written in C++; therefore, it is difficult to extend it to additional instructions or even architectures. Furthermore, it is challenging to reason about the correctness and accuracy of the implementation, which is important as any inaccuracies may lead to bugs being missed in the tested firmware. In order to address these challenges, Chapter 4 proposed binary-level symbolic execution using formal descriptions of ISA semantics. For this purpose, it contributed a formal RISC-V model that is specifically tailored to the creation of custom ISA interpreters. On top of this model, the chapter presented a novel symbolic execution engine which uses the language primitives of the formal model as an abstraction layer, thus achieving extensibility. While this engine is written in Haskell, the chapter also outlined a path towards an integration with SymEx-VP through code generation. Conducted experiments show that, in comparison to prior work, symbolic binary code execution based on formal ISA semantics achieves competitive symbolic execution performance. Furthermore, they resulted in the discovery of bugs in an existing symbolic execution engine, thus illustrating the necessity of correctly implementing the ISA.

Building on the symbolic execution approach developed in the prior chapters, Chapter 5 concerned itself with error detection techniques for embedded firmware. Conceptually, a symbolic execution engine enumerates reachable execution paths based on a specific input source. Additionally, it is necessary to check each executed path for desired properties. Regarding these properties, the chapter focused on the detection of memory corruptions, as the majority of embedded firmware is written in the C programming language which is not memory safe. Due to the lack of protection mechanisms for embedded systems, many memory corruptions do not result in an observable crash at runtime [128]. In order to make them observable, the chapter contributed an integration with HardBound, a technique for achieving memory safety in hardware [60]. As this technique requires firmware instrumentation, the chapter also pursued an alternative direction for detecting a particular kind of memory corruption (stack overflows) without instrumentation. Lastly, the chapter investigated the use of symbolic execution to guide incremental conversions of C source code to safer programming languages, thereby protecting against memory corruptions in a production environment. Experiments conducted with the RIOT [9] operating system uncovered 13 previously unknown and highly critical bugs in parts of RIOT's network stack.

Finally, Chapter 6 applied the contributed symbolic execution techniques to complex IoT applications. Specifically, it focused on testing network protocol implementations provided by existing IoT operating systems. Popular protocols used in this domain are stateful (e.g. MQTT-SN [176]) and hence have a large state space, which causes state explosion issues. In order to mitigate these issues, the chapter contributed heuristics and optimizations to improve input generation for this domain. For this purpose, it relied on manually created protocol specifications written in a novel EDSL, which is based on the Scheme programming language. This allows optimizing input generation for a specific protocol, thereby reducing the input space and ensuring that inputs which reach deeper parts of the source code are generated earlier. Experiments conducted with the Zephyr and RIOT operating systems confirmed the feasibility of the approach for this purpose and resulted in the discovery of three previously unknown bugs. Lastly, to ease the creation of "good" protocol specifications, the chapter proposed incremental refinements of created specifications by visualizing the results of prior symbolic executions runs.

We are confident that the conducted research has been impactful with regard to securing firmware in the embedded domain by uncovering potentially security-critical bugs in such firmware. This is especially evident by the 16 previously unknown bugs that have been found, using the proposed techniques, in different components of the popular IoT operating system RIOT. The following table summarizes all bugs found in this thesis:

Component	**Description**	**Bug Report**
`gnrc_ipv6_nib`	Failing assertion with a SLIP interface	#15171
`gnrc_netif`	Deadlock due to missing mutex unlock	#15221
`sock_dns`	Out-of-bounds read on input buffer	#15345
`udhcpc`	Stack-based buffer overflow	#15353
`uri_parser`	Out-of-bounds read on input buffer	#15927
`uri_parser`	Out-of-bounds read on input buffer	#15930
`clif`	Out-of-bounds read on input buffer	#15945
`clif`	Out-of-bounds read on input buffer	#15947
`gnrc_rpl`	Packed struct cast without prior bounds check	#16018
`gnrc_rpl`	Packed struct cast without prior bounds check	#16062
`gnrc_rpl`	Packed struct cast without prior bounds check	#16085
`riscv_common`	Overflow of the ISR stack	#16395
`riscv_common`	Overflow of the ISR stack	#16448
`asymcute`	Deadlock due to missing mutex unlock	#18289
`gnrc_dhcpv6_client`	Memory corruption during option parsing	#18307
`asymcute`	Type confusion and `NULL` pointer dereference	#18434

7.2. Future Work

This thesis employed the proposed binary-level symbolic execution approach for testing RISC-V binary code. For future work, it would be interesting to investigate its application to additional architectures (e.g. ARM). Doing so would ease using the proposed approach in conjunction with a wider range of firmware images targeting different hardware platforms. The utilization of formal descriptions of ISA semantics (Chapter 4) and SystemC TLM overlays (Chapter 3) should serve as a solid foundation for future work in this direction. Regarding usage of formal ISA descriptions, it is also deemed worthwhile to investigate a closer integration of the error detection techniques presented in Chapter 5 and the formal LibRISCV ISA model from Chapter 4. For example, by formally describing the HardBound mechanism for achieving spatial memory safety in hardware using the language primitives provided by LibRISCV. Prior work on CHERI [206] and Sail [5] has demonstrated that it is possible to describe such hardware features in a formal language. In the context of formal binary-level symbolic execution, this is deemed beneficial as it eases composing HardBound-based error detection with additional techniques proposed in this thesis (e.g. the specification-based symbolic execution approach from Chapter 6). Presently, due to the monolithic architecture of SystemC-based VPs, it is challenging to combine and compose these different techniques in a modular way. Additionally, a formal description of HardBound would also allow the utilization of theorem provers to reason about the correctness of the employed error detection. A necessary prerequisite in this regard is the integration of the LibRISCV ISA model with theorem prover definitions provided by prior work. This endeavor is an interesting direction for future work, as it would allow proofing the correctness of the symbolic instruction semantics implemented on top of LibRISCV. As evident by the bugs found in an existing symbolic execution engine in Chapter 4, correctness of the symbolic instruction semantics is essential, as otherwise the symbolic execution engine may miss bugs in the tested firmware.

Lastly, it would be interesting to further optimize the symbolic execution algorithms implemented in our SymEx-VP symbolic execution engine. Presently, SymEx-VP implements Dynamic Symbolic Execution (DSE) and restarts firmware simulation for each newly generated input. For complex embedded operating systems, this can result in a significant performance penalty as the boot code is re-executed for each new input. Therefore, it would be worthwhile to investigate the use of snapshotting and similar techniques to improve simulation performance. A challenge in this regard is the close integration of SymEx-VP and the SystemC simulation kernel. For our DSE implementation, we implemented an in-process simulation restarting feature for SystemC, which is presently in the process of being integrated into the SystemC reference implementation. In order to implement an effective snapshot mechanism, further changes to the simulation kernel may be necessary. In this regard, it is also deemed worthwhile to extend the symbolic reasoning to the SystemC kernel itself. Presently, only the firmware is symbolically executed; hardware models themselves are not subject to symbolic reasoning. Expanding the symbolic reasoning to include the SystemC models, would allow reasoning about information flow within the models (e.g. when the firmware writes a symbolic value to a hardware register). Prior work has already made use of symbolic execution for testing hardware models [119, 118], but it remains to be seen whether it is feasible to symbolically reason about both firmware and hardware at the same time. Achieving this composition would further expand the completeness and accuracy of the symbolic execution approach proposed in this thesis.

Appendix A.

Acronyms

ADL Architecture Description Language. 108

API Application Programming Interface. 25, 92, 97, 99, 100, 102, 104, 120, 135

AST Abstract Syntax Tree. 101, 102

CoAP Constrained Application Protocol. 39, 157, 185, 186

CoRE Constrained RESTful Environments. 121

CPU Central Processing Unit. 8, 16

CSR Control and Status Register. 29, 133

DHCPv4 Dynamic Host Configuration Protocol version 4. 172, 176

DHCPv6 Dynamic Host Configuration Protocol version 6. 172, 173, 176, 177

DMA Direct Memory Access. 42

DNS Domain Name System. 144

DSE Dynamic Symbolic Execution. 18, 19, 27–29, 81, 88, 93, 157, 161, 162, 164, 192, 193

DSL Domain-Specific Language. 57, 58, 74, 93

EDSL Embedded Domain-Specific Language. 56, 58, 62, 63, 65, 68, 74, 97, 98, 101, 152–154, 156, 158, 159, 166, 168, 170, 171, 179, 180, 191

ELF Executable and Linkable Format. 36, 69, 129, 130

FIFO First In, First Out. 30–32, 47, 48, 50

GADT	Generalized Algebraic Data Type. 84
HAL	Hardware Abstraction Layer. 22, 42
HDL	Hardware Description Language. 22
HTML	HyperText Markup Language. 184
ICMPv6	Internet Control Message Protocol version 6. 36, 38, 144, 154, 155
IEEE	Institute of Electrical and Electronics Engineers. 8, 25
IETF	Internet Engineering Task Force. 35, 171
IoT	Internet of Things. 1, 6, 7, 9, 13, 32, 35, 36, 38–41, 43, 120, 121, 129, 137, 144, 151–153, 157, 159, 160, 166, 171, 172, 174, 176–181, 189–191
IP	Internet Protocol. 35, 121, 145, 151
IPv6	Internet Protocol version 6. 35, 36, 38, 131, 144, 145, 154–158, 172, 179
IR	Intermediate Representation. 42, 78–81, 85, 86, 92–94, 123, 178
ISA	Instruction Set Architecture. 5, 6, 8, 12, 13, 15–17, 19, 23, 54–60, 62–66, 68–70, 72, 74, 76–83, 86, 88, 91–101, 105–109, 189, 190, 192
ISR	Interrupt Service Routine. 132, 133, 191
ISS	Instruction Set Simulator. 16, 23–27, 29, 46, 54, 55, 78, 79, 95, 96, 99, 103–109, 115–118, 127, 161, 190
mDNS	multicast Domain Name System. 157, 158
MMIO	Memory-Mapped Input/Output. 6, 21, 25, 26, 29, 30, 39–41, 43, 46, 118, 156, 161, 172
MMU	Memory Management Unit. 111
MSB	Most Significant Bit. 32, 34, 35
NDP	Neighbor Discovery Protocol. 38, 156, 157
POSIX	Portable Operating System Interface. 4, 21, 178

RISC Reduced Instruction Set Computer. 66

RPL Routing Protocol for Low-Power and Lossy Networks. 121, 123

RTL Register-Transfer Level. 15, 24, 53

RX Receive 'X'. 47, 48, 50

SDK Software Development Kit. 39, 42

SISL Scheme-based Input Specification Language. 153–158, 161, 166, 168

SLIP Serial Line Internet Protocol. 30, 35, 36, 38, 40, 121–123, 145, 157, 171, 172, 180, 191

SMT Satisfiability Modulo Theories. 3, 4, 17–19, 27, 29, 35, 39–41, 57, 75, 83, 90, 91, 115, 152, 153, 157, 162, 179, 181

SP Stack Pointer. 128, 132

SPS State Protocol Server. 170

TCB Thread Control Block. 127, 128, 130–132

TLM Transaction-Level Modeling. 8, 15, 22–26, 29–32, 34, 39, 43–46, 50, 51, 53, 96, 110, 113, 115, 117, 130, 161, 171, 189, 192

TX Transmit 'X'. 48, 50

UART Universal Asynchronous Receiver-Transmitter. 30–33, 35, 36, 38, 40, 43–48, 50, 51, 110, 145, 171

UDP User Datagram Protocol. 40, 131, 158, 166–168, 172, 179

URI Uniform Resource Identifier. 121, 144

VLA Variable Length Array. 129

VP Virtual Prototype. 8, 9, 14, 15, 22–25, 27, 30, 37, 43, 46, 47, 50, 53–55, 95–97, 104, 108, 109, 111, 113, 115, 116, 118, 120, 121, 124, 127, 128, 130–133, 135–137, 161, 189, 192

Appendix B.

Modifications

This published version of the dissertation has been slightly revised. As required by the 2022 promotion regulations of the University of Bremen, this chapter outlines the performed modifications:

- The document has been reformatted for a one-side layout as, contrary to the submitted version, which was optimized for print, it is published as a digital document.
- In the submitted version, the paper “Freer Monads, More Extensible Effects” by Kiselyov et al. was referenced twice in the bibliography; this has now been fixed.
- In Chapter 1, the publication status of the work from Section 4.2 has been updated.
- Additional information on source code and artifacts has been added to Section 4.2.
- The discussion of the evaluation results presented in Section 4.2 has been expanded.
- Subsection 4.2.3 has been expanded to discuss prior work by Goel et al. [73, 74].
- Figure 4.7 has been reformatted slightly to allow a further increase of the font size.
- An appendix outlining the performed modifications has been added to the table of contents. The existing glossary has been subsumed under this new appendix.

Bibliography

[1] Alif Ahmed, Farimah Farahmandi, and Prabhat Mishra. "Directed test generation using concolic testing on RTL models." In: *2018 Design, Automation & Test in Europe Conference & Exhibition*. DATE. 2018, pp. 1538–1543. DOI: 10.23919/DATE.2018.8342260.

[2] Roger Alexander et al. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. Mar. 2012. DOI: 10.17487/RFC6550.

[3] Marco Angelini et al. "SymNav: Visually Assisting Symbolic Execution." In: *2019 IEEE Symposium on Visualization for Cyber Security (VizSec)*. Vancouver, Canada, Oct. 2019, pp. 1–11. DOI: 10.1109/VizSec48167.2019.9161524.

[4] ARM Limited. *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. v8.2 Beta. 2017. URL: https://documentation-service.arm.com/static/5f8ee9d2f86e16515cdbe545.

[5] Alasdair Armstrong et al. "ISA Semantics for ARMv8-a, RISC-V, and CHERI-MIPS." In: *Proceedings of the ACM on Programming Languages*. POPL 3 (Jan. 2019). DOI: 10.1145/3290384.

[6] Hooman Asadian, Paul Fiterău-Broştean, Bengt Jonsson, and Konstantinos Sagonas. "Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification." In: *2022 IEEE Conference on Software Testing, Verification and Validation*. ICST. 2022, pp. 70–81. DOI: 10.1109/ICST53961.2022.00019.

[7] Cornelius Aschermann et al. "NAUTILUS: Fishing for Deep Bugs with Grammars." In: *The Network and Distributed System Security Symposium 2019*. NDSS. San Diego, California, Feb. 2019. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04A-3_Aschermann_paper.pdf.

[8] Thanassis Avgerinos et al. "Automatic Exploit Generation." In: *Communications of the ACM* 57.2 (Feb. 2014), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/2560217.2560219.

[9] Emmanuel Baccelli et al. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT." In: *IEEE Internet of Things Journal.* IoT-J 5.6 (Dec. 2018), pp. 4428–4440. ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2815038.

[10] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques." In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657.

[11] Michael Ballantyne, Alexis King, and Matthias Felleisen. "Macros for Domain-Specific Languages." In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428297.

[12] Greg Banks et al. "SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr." In: *Information Security.* Ed. by Sokratis K. Katsikas et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 343–358. ISBN: 978-3-540-38343-7. DOI: 10.1007/11836810_25.

[13] Sébastien Bardin et al. "The BINCOA Framework for Binary Code Analysis." In: *Computer Aided Verification.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 165–170. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_13.

[14] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories." In: *Handbook of Model Checking.* Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11.

[15] Rob von Behren et al. "Capriccio: Scalable Threads for Internet Services." In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles.* SOSP. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 268–281. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945471.

[16] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference.* ATEC. Anaheim, CA: USENIX Association, 2005, p. 41. URL: https://www.usenix.org/legacy/events/usenix05/tech/freenix/bellard.html.

[17] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. *Uniform Resource Identifier (URI): Generic Syntax.* RFC 3986. Jan. 2005. DOI: 10.17487/RFC3986.

[18] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Berlin, Heidelberg: Springer Berlin Heidelberg, Mar. 2004. ISBN: 978-3-540-20854-9. DOI: 10.1007/978-3-662-07964-5.

[19] Al Bessey et al. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World." In: *Communications of the ACM* 53.2 (Feb. 2010), pp. 66–75. ISSN: 0001-0782. DOI: 10.1145/1646353.1646374.

[20] Surupa Biswas et al. "Memory Overflow Protection for Embedded Systems Using Run-Time Checks, Reuse, and Compression." In: *ACM Trans. Embed. Comput. Syst.* 5.4 (Nov. 2006), pp. 719–752. ISSN: 1539-9087. DOI: 10.1145/1196636.1196637.

[21] David C. Black and Bill Bunton. *SystemC: From the Ground Up.* second. Springer US, 2010. ISBN: 978-0-387-69957-8. DOI: 10.1007/978-0-387-69958-5.

[22] Bluespec, Inc. *Forvis: A Formal RISC-V ISA Specification.* Version 0c5590a. GitHub. Mar. 7, 2020. URL: https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec (visited on 02/01/2024).

[23] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed Greybox Fuzzing." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* CCS. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134020.

[24] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks.* RFC 7228. May 2014. DOI: 10.17487/RFC7228.

[25] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. "Memory models in symbolic execution: key ideas and new thoughts." In: *Software Testing, Verification and Reliability* 29.8 (2019). DOI: 10.1002/stvr.1722.

[26] Thomas Bourgeat et al. "Flexible Instruction-Set Semantics via Abstract Monads (Experience Report)." In: *Proceedings of the ACM on Programming Languages* 7.ICFP (Aug. 2023). DOI: 10.1145/3607833.

[27] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution." In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 234–245. ISSN: 0362-1340. DOI: 10.1145/390016.808445.

[28] Sergey Bratus et al. “Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation.” In: *Usenix ;login:* 36 (2011), pp. 13–21. URL: https://www.usenix.org/system/files/login/articles/105516-Bratus.pdf.

[29] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. “BAP: A Binary Analysis Platform.” In: *Computer Aided Verification.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_37.

[30] Dennis Brylow, Niels Damgaard, and Jens Palsberg. “Static checking of interrupt-driven software.” In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* 2001, pp. 47–56. DOI: 10.1109/ICSE.2001.919080.

[31] Jacob Burnim and Koushik Sen. “Heuristics for Scalable Dynamic Test Generation.” In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* 2008, pp. 443–446. DOI: 10.1109/ASE.2008.69.

[32] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation.* OSDI. San Diego, California: USENIX Association, 2008, pp. 209–224. URL: https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems.

[33] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later.” In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795.

[34] Cristian Cadar et al. “EXE: Automatically Generating Inputs of Death.” In: *ACM Trans. Inf. Syst. Secur.* 12.2 (Dec. 2008). ISSN: 1094-9224. DOI: 10.1145/1455518.1455522.

[35] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. “Device-Agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation.” In: *Annual Computer Security Applications Conference.* ACSAC. Austin, USA: Association for Computing Machinery, 2020, pp. 746–759. ISBN: 978-1-4503-8858-0. DOI: 10.1145/3427228.3427280.

[36] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. "End-to-End Verification of Stack-Space Bounds for C Programs." In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 270–281. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594301.

[37] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware." In: *The Network and Distributed System Security Symposium 2016*. NDSS. San Diego, California, Feb. 2016. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/towards-automated-dynamic-analysis-linux-based-embedded-firmware.pdf.

[38] Jiongyi Chen et al. "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing." In: *The Network and Distributed System Security Symposium 2018*. NDSS. San Diego, California, Feb. 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-1_Chen_paper.pdf.

[39] Xingman Chen et al. "MTSan: A Feasible and Practical Memory Sanitizer for Fuzzing COTS Binaries." In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 841–858. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/chen-xingman.

[40] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, June 2007. ISBN: 978-0-321-42477-8.

[41] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems." In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 265–278. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950396.

[42] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. "Verifying SystemC: A software model checking approach." In: *Formal Methods in Computer Aided Design*. 2010, pp. 51–59. ISBN: 978-0-9835678-0-6.

[43] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. “Bounded Model Checking Using Satisfiability Solving.” In: *Formal Methods in System Design* 19.1 (2001), pp. 7–34. DOI: 10.1023/A:1011276507260.

[44] Edmund M. Clarke and Jeannette M. Wing. “Formal Methods: State of the Art and Future Directions.” In: *ACM Comput. Surv.* 28.4 (Dec. 1996), pp. 626–643. ISSN: 0360-0300. DOI: 10.1145/242223.242257.

[45] Abraham A. Clements et al. “HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation.” In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1201–1218. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/clements.

[46] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. “Prospex: Protocol Specification Extraction.” In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE S&P. 2009, pp. 110–125. DOI: 10.1109/SP.2009.14.

[47] Jeremy Condit et al. “Dependent Types for Low-Level Programming.” In: *Programming Languages and Systems*. Ed. by Rocco De Nicola. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 520–535. ISBN: 978-3-540-71316-6.

[48] Nathan Cooprider et al. “Efficient Memory Safety for TinyOS.” In: *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*. SenSys. Sydney, Australia: ACM, 2007, pp. 205–218. ISBN: 978-1-59593-763-6. DOI: 10.1145/1322263.1322283.

[49] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. “Inception: System-Wide Security Testing of Real-World Embedded Systems Software.” In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 309–326. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani.

[50] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.” In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL. Los Angeles, California: Asso-

ciation for Computing Machinery, 1977, pp. 238–252. ISBN: 978-1-4503-7350-0. DOI: 10.1145/512950.512973.

[51] Enfang Cui, Tianzheng Li, and Qian Wei. "RISC-V Instruction Set Architecture Extensions: A Survey." In: *IEEE Access* 11 (2023), pp. 24696–24711. DOI: 10.1109/ACCESS.2023.3246491.

[52] Pascal Cuoq et al. "Frama-C." In: *Software Engineering and Formal Methods*. Ed. by George Eleftherakis, Mike Hinchey, and Mike Holcombe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 233–247. ISBN: 978-3-642-33826-7.

[53] Robin David et al. "Specification of Concretization and Symbolization Policies in Symbolic Execution." In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA. Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 36–46. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931048.

[54] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution." In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 463–478. ISBN: 978-1-931971-03-4. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson.

[55] Debugging Information Format Committee. *DWARF Debugging Information Format. Version 4*. Tech. rep. Debugging Information Format Committee, 2010. URL: http://www.dwarfstd.org/doc/DWARF4.pdf.

[56] Steve E. Deering and Bob Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. July 2017. DOI: 10.17487/RFC8200.

[57] Ulan Degenbaev. "Formal Specification of the x86 Instruction Set Architecture." PhD thesis. Saarbrücken, Saarland: Universität des Saarlandes, Feb. 2012. DOI: 10.22028/D291-26338.

[58] Design Automation Standards Committee. *IEEE Standard for Verilog Hardware Description Language*. Tech. rep. IEEE, 2006, pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[59] Design Automation Standards Committee. *IEEE Standard for VHDL Language Reference Manual*. Tech. rep. IEEE, 2019, pp. 1–673. DOI: 10.1109/IEEESTD.2019.8938196.

[60] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. "HardBound: Architectural Support for Spatial Safety of the C Programming Language." In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 103–114. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346295.

[61] Adel Djoudi and Sébastien Bardin. "BINSEC: Binary Code Analysis with Low-Level Regions." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 212–217. ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_17.

[62] Ralph Droms. *Dynamic Host Configuration Protocol*. RFC 2131. Mar. 1997. DOI: 10.17487/RFC2131. URL: https://www.rfc-editor.org/info/rfc2131.

[63] Junhan Duan, Yudi Yang, Jie Zhou, and John Criswell. "Refactoring the FreeBSD Kernel with Checked C." In: *2020 IEEE Secure Development (SecDev)*. 2020, pp. 15–22. DOI: 10.1109/SecDev45635.2020.00018.

[64] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. "Contiki - a lightweight and flexible operating system for tiny networked sensors." In: *29th Annual IEEE International Conference on Local Computer Networks*. 2004, pp. 455–462. DOI: 10.1109/LCN.2004.38.

[65] Archibald Samuel Elliott, Andrew Ruef, Micheal Hicks, and David Tarditi. "Checked C: Making C Safe by Extension." In: *2018 IEEE Cybersecurity Development (SecDev)*. Sept. 2018, pp. 53–60. DOI: 10.1109/SecDev.2018.00015.

[66] Andrew Fasano et al. "SoK: Enabling Security Analyses of Embedded Systems via Rehosting." In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS. Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 687–701. ISBN: 978-1-4503-8287-8. DOI: 10.1145/3433210.3453093.

[67] A. Fauth, J. Van Praet, and M. Freericks. "Describing instruction set processors using nML." In: *Proceedings the European Design and Test Conference. ED&TC 1995*. 1995, pp. 503–507. DOI: 10.1109/EDTC.1995.470354.

[68] Bo Feng, Alejandro Mera, and Long Lu. "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling." In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1237–1254. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/feng.

[69] Paul Fiterau-Brostean et al. "Analysis of DTLS Implementations Using Protocol State Fuzzing." In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2523–2540. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[70] Free and Open Source Silicon Foundation. *Embench: A Modern Embedded Benchmark Suite.* URL: https://www.embench.org/ (visited on 01/24/2023).

[71] Galois Inc. *GRIFT - Galois RISC-V ISA Formal Tools.* Version ab2cf5b. GitHub. Aug. 4, 2023. URL: https://github.com/GaloisInc/grift/tree/ab2cf5bd6f2650bdf2fcd6470a4ffdcd4fba6176 (visited on 02/01/2024).

[72] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. "Grammar-Based Whitebox Fuzzing." In: PLDI. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 206–215. ISBN: 978-1-59593-860-2.

[73] Shilpi Goel and Warren A. Hunt. "Automated Code Proofs on a Formal Model of the X86." In: *Verified Software: Theories, Tools, Experiments*. Ed. by Ernie Cohen and Andrey Rybalchenko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 222–241. ISBN: 978-3-642-54108-7. DOI: 10.1007/978-3-642-54108-7_12.

[74] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. "Engineering a Formal, Executable x86 ISA Simulator for Software Verification." In: *Provably Correct Systems*. Ed. by Mike Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog. Cham: Springer International Publishing, 2017, pp. 173–209. ISBN: 978-3-319-48628-4. DOI: 10.1007/978-3-319-48628-4_8.

[75] Dieter Gollmann. *Computer Security*. Third. John Wiley & Sons, Ltd., 2011. ISBN: 978-0-470-74115-3.

[76] Mukesh Gupta and Alex Conta. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification.* RFC 4443. Mar. 2006. DOI: 10.17487/RFC4443.

[77] Eric Gustafson et al. “Toward the Analysis of Embedded Firmware through Automated Re-hosting.” In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019).* Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 135–150. ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/gustafson.

[78] Ali Habibi and Sofiène Tahar. “Design and verification of SystemC transaction-level models.” In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.1 (2006), pp. 57–68. DOI: 10.1109/TVLSI.2005.863187.

[79] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. “Operating Systems for Low-End Devices in the Internet of Things: A Survey.” In: *IEEE Internet of Things Journal.* IoT-J 3.5 (Oct. 2016), pp. 720–734. ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2505901.

[80] Xing Han, Qiaoyan Wen, and Zhao Zhang. “A mutation-based fuzz testing approach for network protocol vulnerability detection.” In: *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology.* 2012, pp. 1018–1022. DOI: 10.1109/ICCSNT.2012.6526099.

[81] Martin Hentschel, Reiner Hähnle, and Richard Bubel. “Visualizing Unbounded Symbolic Execution.” In: *Tests and Proofs.* Ed. by Martina Seidl and Nikolai Tillmann. Cham: Springer International Publishing, 2014, pp. 82–98. ISBN: 978-3-319-09099-3. DOI: 10.1007/978-3-319-09099-3_7.

[82] Vladimir Herdt, Daniel Große, and Rolf Drechsler. *Enhanced Virtual Prototyping. Featuring RISC-V Case Studies.* Springer Cham, Oct. 2021, p. 247. ISBN: 978-3-030-54830-8. DOI: 10.1007/978-3-030-54828-5.

[83] Vladimir Herdt, Daniel Große, and Rolf Drechsler. “Fast and Accurate Performance Evaluation for RISC-V using Virtual Prototypes.” In: *2020 Design, Automation & Test in Europe Conference & Exhibition.* DATE. 2020, pp. 618–621. DOI: 10.23919/DATE48585.2020.9116522.

[84] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. “Early Concolic Testing of Embedded Binaries with Virtual Prototypes: A RISC-V Case Study.” In: *Proceedings of the 56th Annual Design Automation Conference 2019.* DAC. Las Vegas, NV, USA: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3317807.

[85] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler. "RISC-V based virtual prototype: An extensible and configurable platform for the system-level." In: *Journal of Systems Architecture.* JSA 109 (2020). ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2020.101756.

[86] Vladimir Herdt, Hoang M. Le, Daniel Große, and Rolf Drechsler. "Verifying SystemC Using Intermediate Verification Language and Stateful Symbolic Simulation." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.7 (2019), pp. 1359–1372. DOI: 10.1109/TCAD.2018.2846638.

[87] Lars Hermerschmidt, Stephan Kugelmann, and Bernhard Rumpe. "Towards More Security in Data Exchange: Defining Unparsers with Context-Sensitive Encoders for Context-Free Grammars." In: *2015 IEEE Security and Privacy Workshops.* 2015, pp. 134–141. DOI: 10.1109/SPW.2015.29.

[88] Grant Hernandez et al. "FirmUSB: Vetting USB Device Firmware Using Domain Informed Symbolic Execution." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* CCS. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2245–2262. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134050.

[89] David Honfi, Andras Voros, and Zoltan Micskei. "SEViz: A Tool for Visualizing Symbolic Execution." In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation.* ICST. Graz, Austria, Apr. 2015, pp. 1–8. DOI: 10.1109/ICST.2015.7102631.

[90] Alex Horn et al. "Formal co-validation of low-level hardware/software interfaces." In: *2013 Formal Methods in Computer-Aided Design.* 2013, pp. 121–128. DOI: 10.1109/FMCAD.2013.6679400.

[91] Bo-Yuan Huang et al. "Formal Security Verification of Concurrent Firmware in SoCs using Instruction-Level Abstraction for Hardware." In: *2018 55th ACM/ESDA/IEEE Design Automation Conference.* DAC. 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465794.

[92] Paul Hudak. "Building domain-specific embedded languages." In: *ACM Comput. Surv.* 28.4es (Dec. 1996). ISSN: 0360-0300. DOI: 10.1145/242224.242477.

[93] Paul Hudak. "Modular domain specific languages and tools." In: *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203).* 1998, pp. 134–142. DOI: 10.1109/ICSR.1998.685738.

[94] John Hughes. “The design of a pretty-printing library.” In: *Advanced Functional Programming*. Ed. by Johan Jeuring and Erik Meijer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 53–96. ISBN: 978-3-540-49270-2. DOI: 10.1007/3-540-59451-5_3.

[95] IEEE Computer Society and The Open Group. *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7*. Tech. rep. IEEE, 2018, pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[96] International Organization for Standardization. *Programming languages – C*. Standard. Geneva, CH: International Organization for Standardization, Dec. 1999. URL: https://www.iso.org/standard/29237.html.

[97] ITU-T. *Overview of the Internet of things*. ITU-T Y.4000/Y.2060. June 15, 2012. URL: https://handle.itu.int/11.1002/1000/11559.

[98] Samuel Jero et al. “TAG: Tagged Architecture Guide.” In: *ACM Comput. Surv.* 55.6 (Dec. 2022). ISSN: 0360-0300. DOI: 10.1145/3533704.

[99] Trevor Jim et al. “Cyclone: A safe dialect of C.” In: *USENIX 2002 Annual Conference*. Monterey, California: USENIX Association, June 2002, pp. 275–288. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix02/jim.html.

[100] John Aynsley. *OSCI TLM-2.0 Language Reference Manual*. Tech. rep. Open SystemC Initiative (OSCI), July 2009, pp. 1–184. URL: https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf.

[101] Daniel Kästner and Christian Ferdinand. “Proving the Absence of Stack Overflows.” In: *Computer Safety, Reliability, and Security*. Ed. by Andrea Bondavalli and Felicita Di Giandomenico. Cham: Springer International Publishing, 2014, pp. 202–213. ISBN: 978-3-319-10506-2. DOI: 10.1007/978-3-319-10506-2_14.

[102] Gregory Maxwell Kelly. “A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on.” In: *Bulletin of the Australian Mathematical Society* 22.1 (Aug. 1980), pp. 1–83. ISSN: 1755-1633, 0004-9727. DOI: 10.1017/S0004972700006353.

[103] Sang Cheol Kim, Haeyong Kim, JunKeun Song, and Pyeongsoo Mah. “A Dynamic Stack Allocating Method in Multi-Threaded Operating Systems for Wireless Sensor Network Platforms.” In: *2007 IEEE International Symposium on Consumer Electronics*. 2007, pp. 1–6. DOI: 10.1109/ISCE.2007.4382142.

[104] James C. King. “A New Approach to Program Testing.” In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 228–233. ISSN: 0362-1340. DOI: 10.1145/390016.808444.

[105] Oleg Kiselyov and Hiromi Ishii. “Freer Monads, More Extensible Effects.” In: *SIGPLAN Not.* 50.12 (Aug. 2015), pp. 94–105. ISSN: 0362-1340. DOI: 10.1145/2887747.2804319.

[106] Oleg Kiselyov, Amr Sabry, and Cameron Swords. “Extensible Effects An Alternative to Monad Transformers.” In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. Vol. 48. Jan. 2014, pp. 59–70. DOI: 10.1145/2578854.2503791.

[107] Takahisa Kitagawa, Miyuki Hanaoka, and Kenji Kono. “AspFuzz: A state-aware protocol fuzzer based on application-layer protocols.” In: *The IEEE symposium on Computers and Communications*. 2010, pp. 202–208. DOI: 10.1109/ISCC.2010.5546704.

[108] George Klees et al. “Evaluating Fuzz Testing.” In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243804.

[109] Karl Koscher, Tadayoshi Kohno, and David Molnar. “SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems.” In: *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, Aug. 2015. URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher.

[110] Daniel Kroening and Natasha Sharygina. “Formal verification of SystemC by automatic hardware/software partitioning.” In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE ’05.* 2005, pp. 101–110. DOI: 10.1109/MEMCOD.2005.1487900.

[111] Daniel Kroening and Michael Tautschnig. “CBMC – C Bounded Model Checker.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391. ISBN: 978-3-642-54862-8. DOI: 10.1007/978-3-642-54862-8_26.

[112] Martine Lenders et al. “Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things.” In: *Computing Research Repository* abs/1801.02833 (Aug. 2018). DOI: 10.48550/arXiv.1801.02833.

[113] Rainer Leupers et al. “Virtual platforms: Breaking new grounds.” In: *2012 Design, Automation & Test in Europe Conference & Exhibition*. DATE. 2012, pp. 685–690. DOI: 10.1109/DATE.2012.6176558.

[114] Amit Levy et al. “Multiprogramming a 64kB Computer Safely and Efficiently.” In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP. Shanghai, China: ACM, 2017, pp. 234–251. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132786.

[115] Sheng Liang and Paul Hudak. “Modular denotational semantics for compiler construction.” In: *Programming Languages and Systems*. Ed. by Hanne Riis Nielson. ESOP. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 219–234. ISBN: 978-3-540-49942-8. DOI: 10.1007/3-540-61055-3_39.

[116] Sheng Liang, Paul Hudak, and Mark Jones. “Monad Transformers and Modular Interpreters.” In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 333–343. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199528.

[117] Junghee Lim and Thomas Reps. “TSL: A System for Generating Abstract Interpreters and Its Application to Machine-Code Analysis.” In: *ACM Trans. Program. Lang. Syst.* 35.1 (Apr. 2013). ISSN: 0164-0925. DOI: 10.1145/2450136.2450139.

[118] Bin Lin, Zhenkun Yang, Kai Cong, and Fei Xie. “Generating high coverage tests for SystemC designs using symbolic execution.” In: *2016 21st Asia and South Pacific Design Automation Conference*. ASPDAC. 2016, pp. 166–171. DOI: 10.1109/ASPDAC.2016.7428006.

[119] Bin Lin et al. “Concolic testing of SystemC designs.” In: *2018 19th International Symposium on Quality Electronic Design*. ISQED. 2018, pp. 1–7. DOI: 10.1109/ISQED.2018.8357256.

[120] V. Benjamin Livshits and Monica S. Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis.” In: *14th USENIX Security Symposium (USENIX Security 05)*. Baltimore, MD: USENIX Association, Aug. 2005. URL: https://www.usenix.org/legacy/events/sec05/tech/livshits.html.

[121] Peter Marwedel. *Embedded System Design*. Fourth. Cham: Springer International Publishing, 2021, pp. 1–433. ISBN: 978-3-030-60910-8. DOI: 10.1007/978-3-030-60910-8_1.

[122] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. “DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis.” In: *2021 IEEE Symposium on Security and Privacy*. IEEE S&P. 2021, pp. 1938–1954. DOI: 10.1109/SP40001.2021.00018.

[123] Barton P. Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities.” In: *Communications of the ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[124] Barton P. Miller, Louis Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities.” In: *Communications of the ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[125] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[126] Tomek Mrugalski et al. *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*. RFC 8415. Nov. 2018. DOI: 10.17487/RFC8415.

[127] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. “Avatar 2: A multi-target orchestration platform.” In: *The Network and Distributed System Security Symposium 2018*. NDSS. San Diego, California, Feb. 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/07/bar2018_1_Muench_paper.pdf.

[128] Marius Muench et al. "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices." In: *The Network and Distributed System Security Symposium 2018*. NDSS. San Diego, California, Feb. 2018. URL: `https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf`.

[129] Rajdeep Mukherjee, Mitra Purandare, Raphael Polig, and Daniel Kroening. "Formal techniques for effective co-verification of hardware/software co-designs." In: *2017 54th ACM/EDAC/IEEE Design Automation Conference*. DAC. 2017, pp. 1–6. DOI: `10.1145/3061639.3062253`.

[130] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. "WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking." In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO. Orlando, FL, USA: Association for Computing Machinery, 2014, pp. 175–184. ISBN: 978-1-4503-2670-4. DOI: `10.1145/2581122.2544147`.

[131] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C." In: *SIGPLAN Not.* 44.6 (June 2009), pp. 245–258. ISSN: 0362-1340. DOI: `10.1145/1543135.1542504`.

[132] Roberto Natella. "StateAFL: Greybox fuzzing for stateful network servers." In: *Empirical Software Engineering* 27.7 (Oct. 2022), p. 191. ISSN: 1573-7616. DOI: `10.1007/s10664-022-10233-3`.

[133] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation." In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: `10.1145/1250734.1250746`.

[134] Nuno Neves et al. "Using Attack Injection to Discover New Vulnerabilities." In: *International Conference on Dependable Systems and Networks (DSN'06)*. 2006, pp. 457–466. DOI: `10.1109/DSN.2006.72`.

[135] Dorottya Papp, Zhendong Ma, and Levente Buttyan. "Embedded Systems Security: Threats, Vulnerabilities, and Attack Taxonomy." In: *2015 13th Annual Conference on Privacy, Security and Trust*. PST. 2015, pp. 145–152. DOI: `10.1109/PST.2015.7232966`.

[136] Sung Ho Park, Dong Kyu Lee, and Soon Ju Kang. "Compiler-Assisted Maximum Stack Usage Measurement Technique for Efficient Multi-threading in Memory-Limited Embedded Systems." In: *Computers, Networks, Systems, and Industrial Engineering 2011*. Ed. by Roger Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 113–129. ISBN: 978-3-642-21375-5. DOI: `10.1007/978-3-642-21375-5_10`.

[137] Tomsy Paul and G. Santhosh Kumar. "Safe Contiki OS: Type and Memory Safety for Contiki OS." In: *2009 International Conference on Advances in Recent Technologies in Communication and Computing*, pp. 169–171. DOI: `10.1109/ARTCom.2009.126`.

[138] Mathias Payer. "The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes." In: *IEEE Security Privacy* 17.1 (Jan. 2019), pp. 78–82. ISSN: 1558-4046. DOI: `10.1109/MSEC.2018.2889892`.

[139] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "AFLNET: A Greybox Fuzzer for Network Protocols." In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*. ICST. 2020, pp. 460–465. DOI: `10.1109/ICST46399.2020.00062`.

[140] Van-Thuan Pham et al. "Smart Greybox Fuzzing." In: *IEEE Transactions on Software Engineering* 47.9 (2021), pp. 1980–1997. DOI: `10.1109/TSE.2019.2941681`.

[141] Van-Thuan Pham et al. "Smart Greybox Fuzzing." In: *IEEE Transactions on Software Engineering* 47.9 (Sept. 2021). DOI: `10.1109/TSE.2019.2941681`.

[142] Pascal Pieper, Vladimir Herdt, Daniel Große, and Rolf Drechsler. "Dynamic Information Flow Tracking for Embedded Binaries using SystemC-based Virtual Prototypes." In: *2020 57th ACM/IEEE Design Automation Conference*. DAC. 2020, pp. 1–6. DOI: `10.1109/DAC18072.2020.9218494`.

[143] Sonal Pinto and Michael S. Hsiao. "RTL functional test generation using factored concolic execution." In: *2017 IEEE International Test Conference*. ITC. 2017, pp. 1–10. DOI: `10.1109/TEST.2017.8242038`.

[144] Sebastian Poeplau and Aurélien Francillon. "Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and Its Generation." In: *Proceedings of the 35th Annual Computer Security Applications Conference*. AC-

SAC. San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019, pp. 163–176. ISBN: 978-1-4503-7628-0. DOI: 10.1145/3359789.3359796.

[145] Jon Postel. *Internet Protocol.* RFC 791. Sept. 1981. DOI: 10.17487/RFC0791.

[146] Jon Postel. *User Datagram Protocol.* RFC 768. Aug. 1980. DOI: 10.17487/RFC0768.

[147] David A. Ramos and Dawson R. Engler. "Practical, Low-Effort Equivalence Verification of Real Code." In: *Computer Aided Verification.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 669–685. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_55.

[148] John Regehr. "Random Testing of Interrupt-Driven Software." In: *Proceedings of the 5th ACM International Conference on Embedded Software.* EMSOFT. Jersey City, NJ, USA: Association for Computing Machinery, 2005, pp. 290–298. ISBN: 1-59593-091-4. DOI: 10.1145/1086228.1086282.

[149] John Regehr, Alastair Reid, and Kirk Webb. "Eliminating Stack Overflow by Abstract Interpretation." In: *ACM Trans. Embed. Comput. Syst.* 4.4 (Nov. 2005), pp. 751–778. ISSN: 1539-9087. DOI: 10.1145/1113830.1113833.

[150] Alastair Reid. "Trustworthy Specifications of ARM® V8-A and v8-M System Level Architecture." In: *2016 Formal Methods in Computer-Aided Design (FMCAD).* Oct. 2016, pp. 161–168. DOI: 10.1109/FMCAD.2016.7886675.

[151] Sandro Rigo, Guido Araujo, Marcus Bartholomeu, and Rodolfo Azevedo. "ArchC: a SystemC-based architecture description language." In: *16th Symposium on Computer Architecture and High Performance Computing.* 2004, pp. 66–73. DOI: 10.1109/SBAC-PAD.2004.8.

[152] RISC-V Foundation. *ISA Formal Spec Public Review.* Version d42ce01. GitHub. June 18, 2020. URL: https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review (visited on 02/01/2024).

[153] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA.* Ed. by Andrew Waterman and Krste Asanović. Document Version 20191213. Dec. 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[154] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Ed. by Andrew Waterman and Krste Asanović. Document Version 20190608-Priv-MSU-Ratified. June 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf.

[155] John Romkey. *Nonstandard for transmission of IP datagrams over serial lines: SLIP*. RFC 1055. June 1988. DOI: 10.17487/RFC1055.

[156] Andrew Ruef et al. "Achieving Safety Incrementally with Checked C." In: *Principles of Security and Trust*. Ed. by Flemming Nielson and David Sands. Cham: Springer International Publishing, 2019, pp. 76–98. ISBN: 978-3-030-17138-4. DOI: 10.1007/978-3-030-17138-4_4.

[157] Ahmad-Reza Sadeghi, Christan Wachsmann, and Michael Waidner. "Security and Privacy Challenges in Industrial Internet of Things." In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference*. DAC. June 2015, pp. 1–6. DOI: 10.1145/2744769.2747942.

[158] Raimondas Sasnauskas et al. "KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks before Deployment." In: *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN. Stockholm, Sweden: Association for Computing Machinery, 2010, pp. 186–196. ISBN: 978-1-60558-988-6. DOI: 10.1145/1791212.1791235.

[159] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina. "The Halting Problems of Network Stack Insecurity." In: *USENIX ;login:* 36.06 (2011), pp. 22–32. URL: https://www.usenix.org/publications/login/december-2011-volume-36-number-6/halting-problems-network-stack-insecurity.

[160] Tom De Schutter. *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, Mar. 2014. ISBN: 978-1-61730-013-4.

[161] Benjamin Selfridge. "GRIFT: A richly-typed, deeply-embedded RISC-V semantics written in Haskell." In: *SpISA 2019: Workshop on Instruction Set Architecture Specification*. Portland, Oregon, Sept. 2019. URL: https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_10.pdf.

[162] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C.” In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ESEC/FSE. Lisbon, Portugal: Association for Computing Machinery, 2005, pp. 263–272. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081750.

[163] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “AddressSanitizer: A Fast Address Sanity Checker.” In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12).* Boston, MA: USENIX, 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[164] Kostya Serebryany. *OSS-Fuzz - Google's continuous fuzzing service for open source software.* Vancouver, BC, Aug. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany.

[165] Ilya Sergey et al. “Monadic Abstract Interpreters.” In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI. New York, NY, USA: Association for Computing Machinery, June 2013, pp. 399–410. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2491979.

[166] Zach Shelby. *Constrained RESTful Environments (CoRE) Link Format.* RFC 6690. Aug. 2012. DOI: 10.17487/RFC6690.

[167] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP).* RFC 7252. June 2014. DOI: 10.17487/RFC7252.

[168] Alex Shinn, John Cowan, and Arthur A. Gleckler. *Revised^7 Report on the Algorithmic Language Scheme.* Tech. rep. Scheme Language Steering Committee, July 2013. URL: https://small.r7rs.org/attachment/r7rs.pdf.

[169] Yan Shoshitaishvili et al. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis.” In: *2016 IEEE Symposium on Security and Privacy.* IEEE S&P. 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[170] Guoqiang Shu, Yating Hsu, and David Lee. “Detecting Communication Protocol Security Flaws by Formal Fuzz Testing and Machine Learning.” In: *Formal Techniques for Networked and Distributed Systems – FORTE 2008.* Ed. by Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih. Berlin, Hei-

delberg: Springer Berlin Heidelberg, 2008, pp. 299–304. ISBN: 978-3-540-68855-6. DOI: 10.1007/978-3-540-68855-6_19.

[171] SiFive Inc. *SiFive FE310-G000 Manual.* v3p2. Mar. 2021, pp. 1–118. URL: https://sifive.cdn.prismic.io/sifive/4faf3e34-4a42-4c2f-be9e-c77baa4928c7_fe310-g000-manual-v3p2.pdf.

[172] William A. Simpson, Dr. Thomas Narten, Erik Nordmark, and Hesham Soliman. *Neighbor Discovery for IP version 6 (IPv6).* RFC 4861. Sept. 2007. DOI: 10.17487/RFC4861.

[173] JaeSeung Song, Cristian Cadar, and Peter Pietzuch. "SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications." In: *IEEE Transactions on Software Engineering* 40.7 (2014), pp. 695–709. DOI: 10.1109/TSE.2014.2323977.

[174] JaeSeung Song, Tiejun Ma, Cristian Cadar, and Peter Pietzuch. "Rule-Based Verification of Network Protocol Implementations Using Symbolic Execution." In: *2011 Proceedings of 20th International Conference on Computer Communications and Networks.* ICCCN. 2011, pp. 1–8. DOI: 10.1109/ICCCN.2011.6005945.

[175] William Stallings. *Computer Organization and Architecture: Designing for Performance.* Ninth. Pearson Education Inc., 2012. ISBN: 978-0-13-293633-0.

[176] Andy Stanford-Clark and Hong L. Truong. *MQTT For Sensor Networks (MQTT-SN). Protocol Specification.* Standard. Version 1.2. Nov. 14, 2013. URL: http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf.

[177] Guy L. Steele. "Building Interpreters by Composing Monads." In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 472–492. ISBN: 0-89791-636-0. DOI: 10.1145/174675.178068.

[178] Gookwon Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. "Secure Program Execution via Dynamic Information Flow Tracking." In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS. Boston, MA, USA: Association for Computing Machinery, 2004, pp. 85–96. ISBN: 1-58113-804-0. DOI: 10.1145/1024393.1024404.

[179] Wouter Swierstra. “Data Types à la carte.” In: *Journal of Functional Programming* 18.4 (July 2008), pp. 423–436. ISSN: 0956-7968. DOI: 10.1017/S0956796808006758.

[180] System C Standardization Working Group. *IEEE Standard for Standard SystemC Language Reference Manual.* Tech. rep. IEEE, 2012, pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[181] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. “SoK: Eternal War in Memory.” In: *2013 IEEE Symposium on Security and Privacy.* IEEE S&P. May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[182] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. “Model-Based Testing IoT Communication via Active Automata Learning.” In: *2017 IEEE International Conference on Software Testing, Verification and Validation.* ICST. 2017, pp. 276–287. DOI: 10.1109/ICST.2017.32.

[183] Sören Tempel, Tobias Brandt, and Christoph Lüth. *Artifacts for the 2023 Trends in Functional Programming Publication: Versatile and Flexible Modelling of the RISC-V Instruction Set Architecture.* Apr. 2023. DOI: 10.5281/zenodo.7817414.

[184] Sören Tempel, Tobias Brandt, and Christoph Lüth. “Versatile and Flexible Modelling of the RISC-V Instruction Set Architecture.” In: *Trends in Functional Programming.* Ed. by Stephen Chang. Boston, MA, USA: Springer International Publishing, Jan. 2023, pp. 16–35. ISBN: 978-3-031-21314-4. DOI: 10.1007/978-3-031-38938-2_2.

[185] Sören Tempel, Tobias Brandt, Christoph Lüth, and Rolf Drechsler. “Accurate and Extensible Symbolic Execution of Binary Code based on Formal ISA Semantics.” In: *International Conference on Software Engineering and Formal Methods.* SEFM. Aveiro, Portugal, Nov. 2024, pp. 1–18. Under Review.

[186] Sören Tempel, Tobias Brandt, Christoph Lüth, and Rolf Drechsler. *Benchmarks for comparing binary-level symbolic execution speed.* Apr. 2024. DOI: 10.5281/zenodo.10925791.

[187] Sören Tempel, Tobias Brandt, Christoph Lüth, and Rolf Drechsler. “Minimally Invasive Generation of RISC-V Instruction Set Simulators from Formal ISA Models.” In: *2023 Forum on Specification & Design Languages.* FDL. Turin, Italy, Sept. 2023, pp. 1–8. DOI: 10.1109/FDL59689.2023.10272224.

[188] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "An Effective Methodology for Integrating Concolic Testing with SystemC-based Virtual Prototypes." In: *2021 Design, Automation & Test in Europe Conference & Exhibition.* DATE. Grenoble, France, Feb. 2021, pp. 218–221. DOI: 10.23919/DATE51398.2021.9474149.

[189] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. *Artifacts for the 2022 ATVA Paper: SISL: Concolic Testing of Structured Binary Input Formats via Partial Specification.* Zenodo, July 2022. DOI: 10.5281/zenodo.6802198.

[190] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. *Artifacts for the FDL21 Paper: In-Vivo Stack Overflow Detection and Stack Size Estimation for Low-End Multithreaded Operating Systems using Virtual Prototypes.* Sept. 2021. DOI: 10.5281/zenodo.5091709.

[191] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "Automated Detection of Spatial Memory Safety Violations for Constrained Devices." In: *Proceedings of the 27th Asia and South Pacific Design Automation Conference.* ASPDAC. Taipei, Taiwan, Jan. 2022, pp. 160–165. DOI: 10.1109/ASP-DAC52403.2022.9712570.

[192] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "In-Vivo Stack Overflow Detection and Stack Size Estimation for Low-End Multithreaded Operating Systems using Virtual Prototypes." In: *2021 Forum on Specification & Design Languages.* FDL. Antibes, France, Sept. 2021, pp. 1–7. DOI: 10.1109/FDL53530.2021.9568384.

[193] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "SISL: Concolic Testing of Structured Binary Input Formats via Partial Specification." In: *Automated Technology for Verification and Analysis.* Ed. by Ahmed Bouajjani, Holík Lukáš, and Zhilin Wu. ATVA. Beijing, China: Springer International Publishing, Oct. 2022, pp. 77–82. ISBN: 978-3-031-19992-9. DOI: 10.1007/978-3-031-19992-9_5.

[194] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "Specification-Based Symbolic Execution for Stateful Network Protocol Implementations in IoT." In: *IEEE Internet of Things Journal.* IoT-J 10.11 (Jan. 2023), pp. 9544–9555. DOI: 10.1109/JIOT.2023.3236694.

[195] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "SymEx-VP: An Open Source Virtual Prototype for OS-agnostic Concolic Testing of IoT Firmware." In: *Journal*

of Systems Architecture. JSA (May 2022), pp. 1–12. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2022.102456.

[196] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. “Towards Quantification and Visualization of the Effects of Concretization During Concolic Testing.” In: *IEEE Embedded Systems Letters.* ESL 14.4 (Dec. 2022), pp. 195–198. DOI: 10.1109/LES.2022.3171603.

[197] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. “Towards Reliable Spatial Memory Safety for Embedded Software by Combining Checked C with Concolic Testing.” In: *2021 58th ACM/IEEE Design Automation Conference.* DAC. San Francisco, California, Dec. 2021, pp. 667–672. DOI: 10.1109/DAC18074.2021.9586170.

[198] Sören Tempel, Herdt Vladimir, and Rolf Drechsler. *Artifacts for the IEEE Internet of Things Journal Publication: Specification-based Symbolic Execution for Stateful Network Protocol Implementations in the IoT.* Zenodo, Jan. 2023. DOI: 10.5281/zenodo.7515748.

[199] Sören Tempel, Herdt Vladimir, and Rolf Drechsler. *SymEx-VP: An open source virtual prototype for OS-agnostic concolic testing of IoT firmware.* Code Ocean, Apr. 2022. DOI: 10.24433/CO.7255660.v1.

[200] University of California. *Spike, a RISC-V ISA Simulator.* Version 7c89063. GitHub. Jan. 24, 2024. URL: https://github.com/riscv/riscv-isa-sim (visited on 02/01/2024).

[201] Philip Wadler. “A prettier printer.” In: Jeremy Gibbons and Oege de Moor. *The Fun of Programming.* Palgrave, Mar. 2003. ISBN: 0-333-99285-7.

[202] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. “Superion: Grammar-Aware Greybox Fuzzing.” In: *2019 IEEE/ACM 41st International Conference on Software Engineering.* 2019. DOI: 10.1109/ICSE.2019.00081.

[203] Robert N. M. Watson et al. *Arm Morello Programme: Architectural security goals and known limitations.* Tech. rep. UCAM-CL-TR-982. University of Cambridge, Computer Laboratory, July 2023. DOI: 10.48456/tr-982.

[204] Jos Wetzels. “Internet of Pwnable Things: Challenges in Embedded Binary Security.” In: *USENIX ;login:* 42.02 (2017), pp. 73–77. URL: https://www.usenix.org/publications/login/summer2017/wetzels.

[205] Emmett Witchel, Josh Cates, and Krste Asanović. "Mondrian Memory Protection." In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. San Jose, California: Association for Computing Machinery, 2002, pp. 304–316. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605429.

[206] Jonathan Woodruff et al. "The CHERI capability model: Revisiting RISC in an age of risk." In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture*. ISCA. 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201.

[207] Christopher Wright et al. "Challenges in Firmware Re-Hosting, Emulation, and Analysis." In: *ACM Comput. Surv.* 54.1 (Jan. 2021). ISSN: 0360-0300. DOI: 10.1145/3423167.

[208] Hongwei Xi, Chiyan Chen, and Gang Chen. "Guarded Recursive Datatype Constructors." In: *SIGPLAN Not.* 38.1 (Jan. 2003), pp. 224–235. ISSN: 0362-1340. DOI: 10.1145/640128.604150.

[209] Insu Yun et al. "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing." In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[210] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares." In: *The Network and Distributed System Security Symposium 2014*. NDSS. San Diego, California, Feb. 2014. URL: https://www.ndss-symposium.org/ndss2014/programme/avatar-framework-support-dynamic-security-analysis-embedded-systems-firmwares/.

[211] Rui Zhang et al. "An Improved RTEMS Supporting Real-Time Detection of Stack Overflow." In: *Wireless and Satellite Systems*. Ed. by Min Jia, Qing Guo, and Weixiao Meng. Cham: Springer International Publishing, 2019, pp. 283–293. ISBN: 978-3-030-19153-5. DOI: 10.1007/978-3-030-19153-5_29.

[212] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. "LISA-machine description language and generic machine model for HW/SW co-design." In: *VLSI Signal Processing, IX*. 1996, pp. 127–136. DOI: 10.1109/VLSISP.1996.558311.