Towards Reliable Spatial Memory Safety for Embedded Software by Combining Checked C with Concolic Testing

Sören Tempel[1] Vladimir Herdt[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
tempel@uni-bremen.de, vherdt@uni-bremen.de, drechsler@uni-bremen.de

***Abstract*—In this paper we propose to combine the safe C dialect Checked C with concolic testing to obtain an effective methodology for attaining safer C code. Checked C is a modern and backward compatible extension to the C programming language which provides facilities for writing memory-safe C code. We utilize incremental conversions of unsafe C software to Checked C. After each increment, we leverage concolic testing, an effective test generation technique, to support the conversion process by searching for newly introduced and existing bugs.**

Our RISC-V experiments using the RIOT *Operating System* (OS) demonstrate the effectiveness of our approach. We uncovered 4 previously unknown bugs and 3 bugs accidentally introduced through our conversion process.

***Index Terms*—Checked C, Concolic Testing, Memory Safety, Embedded Software, Virtual Prototype, RISC-V, RIOT**

I. Introduction

With the emergence of the Internet of Things (IoT), heavily constrained embedded devices are being deployed in security critical areas. As per RFC 7228, these devices are severely constrained in terms of available resources (e.g. available memory or computing power) compared to conventional devices (such as laptops) [1]. For this reason, efficient use of resources is important in the low-end IoT. Unfortunately, many security techniques known from conventional devices (e.g. ASLR) impact these resources and are thus not commonly available on constrained embedded devices. Language-based security techniques are considered a mitigation for this pitfall [2]. Unfortunately, the unsafe programming language C remains a popular choice for heavily constrained embedded devices [3, Table 1]. Safe C dialects attempt to add language-based security features to the C programming language to address this issue. We will refer to standard C as *legacy C* in the following.

A recent advance in regard to safe C dialects is Checked C [4] which is under ongoing development by Microsoft and provides facilities for writing memory-safe C code. These facilities include new syntactic constructs and thus a custom compiler is required for Checked C code. Additionally, existing *Software* (SW) written in legacy C needs to be converted to Checked C to make use of these constructs, e.g. for annotating pointer bounds. This requirement for modifications is a commonly cited drawback of safe C dialects. However, compared to other safe C dialects, Checked C is a fully backward compatible extension of the C programming language. This has the advantage that 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 easing 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 [5, p. 78]. As such, the conversion process relies on manual effort which is susceptible to errors and thus requires cross-validations. We refer to errors introduced during the conversion process as conversion bugs. While these bugs do not lead to potentially exploitable undefined behavior they may still result in spurious crashes, thereby impacting reliability.

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. Recent advances in concolic testing have made it possible to partially automate unit testing [6]. As concolic testing attempts to find new paths based on collected path constraints, we consider this technique promising for finding paths to failing bounds checks which were inserted by the compiler of Checked C.

Contribution: In order to address conversion bugs, we propose to combine Checked C with concolic testing to obtain an effective methodology for finding paths to failing Checked C bounds checks. This combination is capable of uncovering real memory safety violations which would lead to potentially exploitable undefined behavior without Checked C and conversion bugs[1] which impact reliability in a production environment. We also discuss the benefits and limitations of this combination. On the implementation side, we leverage *Virtual Prototypes* (VPs) as an accurate simulation environment for embedded SW binaries and integrate a *Concolic Testing Engine* (CTE) with the VP-based simulation. We refer to this combination as VP-CTE[2]. Our RISC-V based experiments using the RIOT *Operating System* (OS) demonstrate the effectiveness of our approach. Concolic testing has been beneficial in safeguarding the Checked C conversions of existing RIOT code and also allowed us to find 4 previously unknown bugs in the network stack of RIOT.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW19001.

[1]Conversion bugs are further classified and discussed in Section IV-B.

[2]Visit http://www.systemc-verification.org/risc-v to find our most recent RISC-V related approaches.

II. Related Work

Checked C has previously been applied to conventional operating systems, such as FreeBSD [7]. We are also aware of prior work presenting an optional integration of Checked C for RIOT, allowing compilation of converted code with both legacy C and Checked C compilers [8]. We consider these two approaches complementary to our own since they do not address conversion bugs.

Prior work has also presented alternative safe C dialects such as Cyclone [9] and Deputy [10]. 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. [11], [12]. 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 [4]. 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.

Concolic testing is a technique that has been applied in different domains for verification purposes including intermediate representations [13] and binary level SW [14]. In this paper we utilize concolic testing in combination with Checked C.

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.

III. Background

The following subsections serve as a brief primer on Checked C and concolic testing as a SW testing technique.

A. 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. For Checked C developers memory safety has two aspects [4, p. 53]:

> *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.

Checked C only supports the latter. 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 type `_Array_ptr` which must be associated with pointer bounds. Bounds are specified through so-called *bounds expressions* which describe the memory range that can be accessed through a given variable [4].

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"* [4, p. 53], i.e. no spatial safety violations can arise due to code marked as a checked regions. Since Checked C extends legacy C with new syntactic constructs, a custom compiler is required. We utilize the clang-based reference from GitHub[3].

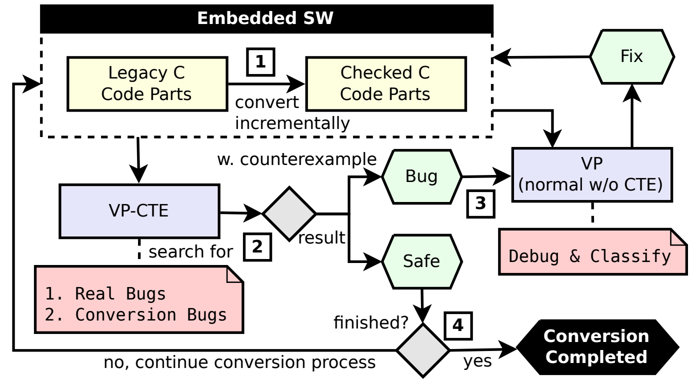

Fig. 1: Overview on our methodology.

B. Concolic Testing

Concolic testing is an automated SW testing technique that continuously generates inputs to explore new paths through the SW program. It works by tracking (symbolic) constraints alongside the concrete execution. An SMT solver is utilized to solve these constraints and thus generate new inputs. Based on an exploration strategy the next input is selected and executed. The exploration process continues until all paths have been explored, some predefined coverage goal has been reached, or an error is detected.

A standard approach for concolic testing is to implement a *Dynamic Symbolic Execution* (DSE) methodology, which randomly flips branch condition constraints (collected during execution) to generate new inputs. Concolic testing also allows to apply concretization, i.e. fixing a symbolic expression to a concrete value, to simplify the reasoning process. A common strategy is to employ *Address Concretization* (AC) in case of symbolic memory indices. Due to concretization strategies, concolic testing is usually unable to provide complete safety proofs. DSE and AC are further described in a publication by Baldoni et al. [15].

IV. Methodology

This section presents our methodology on combining Checked C with concolic testing to attain more reliable spatial memory safety for embedded SW. Fig. 1 provides an overview of the methodology. The starting point is an embedded SW written in legacy C that is converted to Checked C (box 1 in Fig. 1, top). We expect this process to occur incrementally based on logical software modules, as a full conversion at once is deemed difficult and may not be achievable. For this reason, the embedded SW will typically consist of both Checked C and legacy C code parts. After each increment the newly converted module is tested using VP-CTE (box 2 in Fig. 1, left). Essentially, VP-CTE follows a standard concolic testing methodology (as described in Section III-B) at the SW binary level to search for inputs that will cause an execution error (e.g. a failed Checked C bounds check). Additional implementation details will be described in Section V.

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

This approach enables us to discover two kinds of bugs in the (partially) converted SW binary:

1) Real spatial memory safety violations which were already present in the unmodified original legacy C code base and can result in undefined behavior without Checked C.
2) Conversion bugs which were introduced when converting the legacy C code base to Checked C, i.e. crashes which 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 VP-CTE returns a counterexample, i.e. a test case to reproduce the bug (box 3 in Fig. 1, right). Based on the test case, a debugging process can be started using the normal VP (which provides comprehensive debugging capabilities). This allows developers to classify the bug accordingly and provide a fix for the embedded SW.

The conversion process continues incrementally until no more bugs are found and all required SW modules have been converted (box 4 in Fig. 1, bottom).

In the following, we present an example to illustrate the conversion process and conversion bugs (Section IV-A), then we provide a classification of conversion bugs (Section IV-B).

A. Conversion Bug Example

The central 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 [16]. The following example will instead 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 Fig. 2. The figure contains both, the original legacy C implementation (on the left-hand side of Fig. 2) 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 2 using the `_Checked` keyword. Both implementations start by skipping given input until a period or comma character is encountered (Line 7 - Line 11). If no such character is found, `false` is returned (Line 12 - Line 13). Afterwards, `key` (Line 16) and `value` (Line 17) 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 20 to determine whether the extracted value should be further processed with the `parse_period` or the `parse_comma` function.

The legacy C code in Fig. 2 does not contain any spatial memory safety violations. However, during the conversion to Checked C a conversion bug was introduced: The code in Line 20 for accessing the separation character causes a bounds violation as `value - 1` is one character outside of the specified bounds 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 20) 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 spurious crash can be fixed either by widening the pointer bounds of the `value` variable or by accessing the separation character as `*buf` in Line 20.

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 spurious crashes.

B. 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 Section IV-A).
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. Through so-called subsumption checks Checked C ensures 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 [4, p. 57].
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 [4, p. 57].

In this paper 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 Figure 2 narrows the bounds of the `input` variable. Detection of narrowing bugs is ultimately concerned with finding a path where an access, which violates the defined pointer bounds, is performed.

```c
1 bool parse(char *input, size_t len)
2 {
3   char *end = input + len;
4   char *buf = input;
5
6   // Advance buf til separation character is found
7   while (buf < end) {
8     if (*buf == '.' || *buf == ',')
9       break;
10    buf++;
11  }
12  if (buf == end)
13    return false;
14
15  // Extract key and value relative to buf
16  char *key = input;
17  char *value = buf + 1;
18
19  // Consult separation character for further parsing
20  if (*(value - 1) == '.')
21    return parse_period(/* ... */);
22  else
23    return parse_comma(/* ... */);
24 }
```

```c
1 bool parse(_Array_ptr<char> input : count(len), size_t len)
2 _Checked {
3   _Array_ptr<char> end = input + len;
4   _Array_ptr<char> buf : bounds(input, end) = input;
5
6   // Advance buf til separation character is found
7   while (buf < end) {
8     if (*buf == '.' || *buf == ',')
9       break;
10    buf++;
11  }
12  if (buf == end)
13    return false;
14
15  // Extract key and value relative to buf
16  _Array_ptr<char> key : bounds(input, buf) = input;
17  _Array_ptr<char> value : bounds(buf + 1, end) = buf + 1;
18
19  // Consult separation character for further parsing
20  if (*(value - 1) == '.')
21    return parse_period(/* ... */);
22  else
23    return parse_comma(/* ... */);
24 }
```

Fig. 2: Implementation of an exemplary input parser in legacy C (left side) and Checked C (right side).

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 SW. As such, detection of narrow bugs is no different from detection of real spatial memory bugs. They only differ in regards to the bug source. Contrary to real bugs, narrowing bugs are introduced during the conversion process and cannot occur in the original SW. By combining Checked C with concolic testing, we can detect both. We further distinguish and demonstrate these two use-cases in Section V. We leave the detection of functional bugs for future work, but we believe that concolic testing is a suitable foundation for this use-case. This issue is further discussed in Section VI.

V. Evaluation

We evaluate our described approach by applying it to the low-end IoT OS RIOT[4], which is considered by Hahm et al. as the *"most prominent open source OS"* in this domain [3, p. 732]. Apart from being open source, RIOT also provides a modular software architecture, thereby allowing an incremental conversion of the existing source code to Checked C on a per-module basis. Moreover, RIOT employs a code quality management process thereby putting a strong emphasis on code quality and thorough testing [17, p. 4438]. The OS provides an extensive feature set and support for different architectures. As a case-study we consider the RISC-V architecture. RIOT itself is further described in a publication by Baccelli et al. [17].

We implemented VP-CTE by integrating a CTE with the open source RISC-V VP [18] which is available on GitHub[5]. The CTE itself implements standard concolic testing using the DSE and AC techniques (see Section III-B). We have already described the VP-CTE implementation in prior work [19]. This prior work executes arbitrary RISC-V instructions symbolically, thereby allowing us to adapt it for concolic testing of compiled Checked C software. For the evaluation we use the HiFive1 platform of the RISC-V VP, which corresponds to the HiFive1 board from SiFive and is supported by RIOT.

In the following we provide more details on our test setup (Section V-A) and present our obtained results (Section V-B).

A. 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 spatial violations such as buffer overflows (which potentially allow for remote code execution) [20]. In total we incrementally converted 53 functions in 6 different RIOT modules.

We employed a two step methodology for testing the incremental conversion process using VP-CTE. In a first step we created specialized unit tests which target specific functions and modules. Essentially, we therefore created a respective test driver (which is a piece of C code) that calls the software under test with symbolic input values. In a next step we utilized the existing RIOT applications for more extensive integration testing by introducing symbolic input values directly into the network stack using the SLIP (*Serial Line Internet Protocol*) interface. SLIP is a network protocol for the transmission of IP packets over a serial line. In our setup we pack symbolic data into an IPv6 packet at the VP-CTE side, encapsulate it in a SLIP packet, and pass it through a UART into the RIOT SLIP network driver. The driver unpacks the IPv6 packet and forwards it to the network stack. Since the RIOT network stack awaits new packets indefinitely, we added a switch to ensure termination after one packet has been processed.

We register a custom abort handler in RIOT to notify VP-CTE 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 VP-CTE, we can create a test case to reproduce and debug the source of an error, and thereby classify the bug kind (i.e. real

[4]https://www.riot-os.org/

[5]https://github.com/agra-uni-bremen/riscv-vp

TABLE I: Real (spatial) memory bugs (M1-M2), real logic bugs (L1-L2) and conversion bugs (C1-C3) that we found in different RIOT components.

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

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 [21] or AddressSanitizer [22]) 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.

B. Results

Table I lists all errors that we have found in network-related RIOT modules. It has five columns that show in order: 1) the bug id (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 VP-CTE until the bug was found (column: #Path), and 5) the overall execution time in seconds of VP-CTE 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 in 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. VP-CTE has also been beneficial to support the Checked C conversion procedure. Despite being careful in the conversion process we detected three conversion bugs using VP-CTE. We would like to point out that, due to the manual conversion process, such kind of bugs can be easily added by accident. Undetected, they would result in a runtime check failure and thus abort the embedded software application thereby impacting reliability. Table II provides a more detailed description on each of the 7 bugs, for real bugs it also contains a reference to the corresponding issue in the RIOT bug tracker[6].

VI. Discussion and Future Work

Our experiments demonstrate the effectiveness and potential of our approach in combining Checked C with concolic testing to provide more reliable spatial memory safety for real-world embedded SW applications. Nonetheless, there is still room for improvement which we discuss in the following.

Due to concretization strategies, concolic testing is generally unable to guarantee the absence of errors because the complete state space is not fully covered. Therefore, it is rather a bug hunting technique. While our experiments demonstrated its capabilities in this regard, we cannot prove 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. To further improve reliability, we plan to investigate complete proof techniques, e.g. by avoiding concretization in the CTE.

Another interesting research direction is to optimize the number and placement of runtime checks generated by Checked C 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 [4, p. 53]. We envision to utilize concolic testing (and potentially other formal techniques) to prove that certain runtime checks are not necessary (concolic testing can provide such a proof in case no concretization occurs). 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 concolic testing is a suitable foundation to develop effective bug hunting techniques to find such bugs efficiently as well. It can be complemented with complete proof techniques to enable equivalence proofs of the converted Checked C SW with the legacy C SW. While prior work has already utilized symbolic execution for the purpose of equivalence checking, limitations with regard to scalability still remain [23]. 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.

VII. Conclusion

In this paper we proposed to combine Checked C with concolic testing to attain more reliable spatial memory safety for embedded SW. We employ concolic testing to safeguard the incremental conversion process from legacy C to Checked C. Beside conversion bugs, our approach enables to detect spatial memory bugs which have been present in the original

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

TABLE II: More detailed description of all real- and conversion bugs which we found with 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 Section IV-A. **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.

embedded SW. The effectiveness of our approach was demonstrated by applying it to real-world RIOT code for the RISC-V architecture. We found 4 previously unknown bugs in the RIOT network stack, which have been confirmed and fixed by RIOT developers, and 3 conversion bugs added by accident which otherwise would have caused a spurious runtime check failure.

REFERENCES

[1] C. Bormann, M. Ersue, and A. Keränen, "Terminology for Constrained-Node Networks," RFC 7228, May 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7228.txt

[2] J. Wetzels, "Internet of Pwnable Things: Challenges in Embedded Binary Security," *USENIX ;login:*, vol. 42, no. 02, pp. 73–77, 2017. [Online]. Available: https://www.usenix.org/publications/login/summer2017/wetzels

[3] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, Oct. 2016.

[4] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked C: Making C Safe by Extension," in *2018 IEEE Cybersecurity Development (SecDev)*, Sep. 2018, pp. 53–60.

[5] A. Ruef, L. Lampropoulos, I. Sweet, D. Tarditi, and M. Hicks, "Achieving Safety Incrementally with Checked C," in *Principles of Security and Trust*, F. Nielson and D. Sands, Eds. Cham: Springer International Publishing, 2019, pp. 76–98.

[6] K. Sen, D. Marinov, and G. 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*, ser. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, 2005, p. 263–272.

[7] J. Duan, Y. Yang, J. Zhou, and J. Criswell, "Refactoring the FreeBSD Kernel with Checked C," in *2020 IEEE Secure Development (SecDev)*, 2020, pp. 15–22.

[8] S. Tempel and T. Bruns, "RIOT-POLICE: An implementation of spatial memory safety for the RIOT operating system," *arXiv e-prints*, Sep. 2018. [Online]. Available: https://arxiv.org/abs/2005.09516

[9] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX 2002 Annual Conference*. USA: USENIX Association, Jun. 2002, pp. 275–288. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix02/jim.html

[10] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent Types for Low-Level Programming," in *Programming Languages and Systems*, R. De Nicola, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 520–535.

[11] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 245–258.

[12] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hard-Bound: 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*, ser. ASPLOS XIII. New York, NY, USA: Association for Computing Machinery, 2008, p. 103–114.

[13] C. Cadar, D. Dunbar, and D. 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*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[14] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[15] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018.

[16] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 48–62.

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

[18] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101756, 2020.

[19] S. Tempel, V. Herdt, and R. Drechsler, "An Effective Methodology for Integrating Concolic Testing with SystemC-based Virtual Prototypes," in *Design, Automation and Test in Europe Conference (DATE). Design, Automation & Test in Europe (DATE-2021)*, Feb. 2021.

[20] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina, "The Halting Problems of Network Stack Insecurity," *USENIX ;login:*, vol. 36, no. 06, pp. 22–32, 2011. [Online]. Available: https://www.usenix.org/publications/login/december-2011-volume-36-number-6/halting-problems-network-stack-insecurity

[21] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 89–100.

[22] K. Serebryany, D. Bruening, A. Potapenko, and D. 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. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[23] D. A. Ramos and D. R. Engler, "Practical, Low-Effort Equivalence Verification of Real Code," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 669–685.