ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/382214314

Assured Micropatching of Race Conditions in Legacy Real-time Embedded Systems

Conference Paper · July 2024

CITATION
1

READS
63

5 authors, including:

Rik Chatterjee
Colorado State University
5 PUBLICATIONS **12** CITATIONS

SEE PROFILE

All content following this page was uploaded by Rik Chatterjee on 12 July 2024.

The user has requested enhancement of the downloaded file.

Assured Micropatching of Race Conditions in Legacy Real-time Embedded Systems

Rik Chatterjee
Colorado State University
rik.chatterjee@colostate.edu

Ben Karel
Aarno Labs
bkarel@aarno-labs.com

Ricardo Baratto
Aarno Labs
ricardo@aarno-labs.com

Michael Gordon
Aarno Labs
mgordon@aarno-labs.com

Jeremy Daily
Colorado State University
jeremy.daily@colostate.edu

***Abstract*—Embedded systems are inherently event-driven, relying extensively on interrupts to facilitate real-time interactions with hardware. Interrupt-oriented programming is fundamental to the design and functionality of embedded systems, enabling them to dynamically respond to real-time events. Despite careful development efforts, frequent and event-triggered nature of interrupt service routines (ISRs) can still precipitate race conditions. This leads to concurrency vulnerabilities between application tasks and interrupt handlers.**

Existing methods to patch race conditions typically rely on source code, which may not be available in many real-world scenarios. Additionally, traditional patching methods often fail to ensure patches do not interfere with the baseline functionality of the system. Furthermore, compiler optimizations may reorder operations and potentially induce race conditions even when the source code appears safe.

In this paper, we discuss a race condition identified during the testing of an Intrusion Detection and Prevention System (IDPS) developed for research on a Controller Area Network (CAN) used in commercial vehicle systems. We detail the discovery, analysis, and resolution of this race condition through the open-source CodeHawk Binary Patcher, a novel, assured binary micropatching platform. Micropatching is specifically chosen for its ability to change the fewest possible bytes in the system's firmware, thereby minimizing potential side effects while providing validations that the patch preserves the original baseline functionality of the system. This approach is particularly crucial in scenarios where source code is unavailable. Our method not only ensures the effectiveness of the patch but also provides a rigorous assurance case demonstrating that the patches do not interfere with the system's baseline functionality, thus preserving the integrity and validation efforts of the original software. This research not only offers a practical solution to a specific concurrency vulnerability but also significantly enhances the broader framework for addressing race conditions in embedded real-time systems.

I. Introduction

Embedded real-time systems are the backbone of modern automotive technology, driving advancements in vehicle performance, safety, and user experience while enabling a broad spectrum of functionalities. These systems are resource-constrained yet need to operate in real-time, hence rely extensively on interrupts to manage interactions with hardware. Interrupts, essential for concurrency and communication via interrupt service routines (ISRs), are triggered by specific events. While this architecture is crucial for achieving real-time responsiveness and precision, it also introduces significant vulnerabilities, particularly race conditions. These arise from the frequent and nondeterministic interactions of interrupts with application tasks, leading to data races that are challenging to detect, isolate, and correct.

Race conditions in embedded systems can lead to unpredictable behaviors and severe system failures when multiple concurrent execution entities—such as threads, tasks, or interrupts—simultaneously access the same memory location with at least one operation being a write. The resulting uncertainty in access order has led to notable disasters. These include the Therac-25 radiation therapy machine incident, where race conditions in the control software contributed to fatal overdoses [1], the Mars Pathfinder spacecraft's system resets caused by priority inversion [2], and the 2003 blackout across the USA and Canada, where similar conditions exacerbated system failures [3]. Such examples highlight the critical challenge of managing race conditions, which often manifest under specific environmental conditions and during unique execution interleavings, complicating diagnostics and mitigation efforts.

Efforts to mitigate race conditions in embedded systems have historically relied on a variety of techniques, including static and dynamic analysis tools and sophisticated automated repair methods [4], [5], [6]. However, these techniques not only require access to source code and extensive system knowledge but also often fail to provide assurance that modifications will not interfere with the system's baseline functionality. This limitation is particularly critical in legacy environments where source code is unavailable or the system's complexity makes traditional mitigation approaches prohibitive.

In addressing the deficiencies of prevailing race condition mitigation strategies, this study delves into a race condition detected during the evaluation of an Intrusion Detection and Prevention System (IDPS) developed for research in commercial vehicle systems [7]. To resolve this specific issue without access to the original source, we employed the CodeHawk Binary Patcher (CBP) [8], an open-source platform for producing high-assurance micropatches on stripped binaries, without requiring source code nor recompilation. CBP reduces

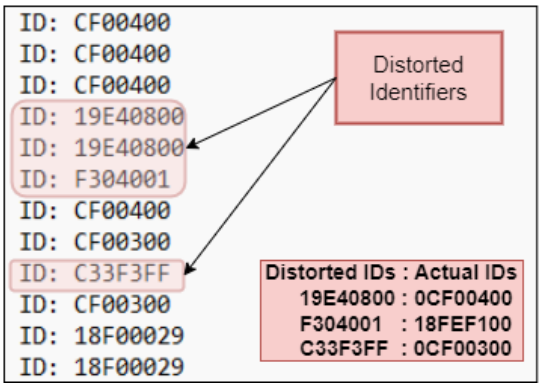

Fig. 1: Distorted CAN Identifiers

the cost of binary patching by lifting functions of a binary into editable C code. After a C developer makes the desired changes directly on the lifting, CBP automatically applies the changed semantics to the binary, with minimal binary changes. Finally, CBP produces a suite of assurance analysis results that provide strong evidence as to whether the patch was applied correctly, and for important classes of bugs, whether the patch fixes the underlying issues.

The strength of CBP comes from modifying minimal binary data to mitigate bugs without extensive system overhaul. CBP was chosen for its capability to make precise alterations that are verifiably non-intrusive to the system's existing sequential functionality, thus maintaining the integrity of previously validated behaviors. The platform not only successfully rectified the presented race condition but also enabled the provably-correct fix of a memory overwrite vulnerability in the IDPS, demonstrating CBP's potential as a crucial tool in the enhancement of embedded system security.

The rest of the paper is organized as follows: Section II presents a brief overview of concurrency vulnerabilities. Section III details the specific race condition identified within the IDPS. Section IV describes our micropatching approach, including the application of CBP to address the identified vulnerability. Section V discusses the broader implications of our findings. Finally, Section VI concludes the paper with final remarks.

II. Background

Race conditions arise when two or more execution threads, or interrupt service routines (ISRs), access shared resources concurrently, and at least one thread modifies the resource. In bare-metal embedded systems, where tasks and ISRs directly interact with hardware without an intermediary operating system, these conditions can lead to unpredictable outcomes. This occurs because the precise order of execution for these concurrent processes is not deterministic and lacks atomicity, making it possible for operations to interfere with one another. Such interruptions can result in data corruption or erratic system behavior, as operations on shared resources are not completed in a single, indivisible sequence.

Historically, the concept and challenges of race conditions have been extensively studied. Netzer and Miller's seminal work [9] provided a formal definition and explored the issues related to race conditions in parallel programs, setting a foundation for subsequent research in the field. Their analysis highlighted the difficulties in both detecting and resolving these conditions, particularly in environments where task interleaving is complex and unpredictable.

Efforts to address race conditions have traditionally involved static and dynamic analysis tools designed to detect potential concurrency issues before they manifest at runtime. Studies like those by Karam et al. [10] and Allen and Padua [11] represent early attempts to tackle race conditions through software engineering approaches focused on Ada and Fortran programming environments, respectively. However, these and other conventional methods often rely on the availability of complete source code and assume a controlled execution environment—conditions that do not typically hold in legacy bare-metal embedded systems.

III. Problem Statement

A. Detailed System Overview of IDPS

TruckSentry was developed as a real-time reactive IDPS for medium and heavy-duty vehicles operating on SAE-J1939 protocols [7]. The system implemented on a prototype with an ARM Cortex M7, continuously monitors network data, assessing them against predefined rules. In cases where messages violate these rules, TruckSentry modifies specific bits within the message to trigger an error frame, thereby preventing the transmission of potentially harmful data. TruckSentry utilizes Non-Return-to-Zero (NRZ) encoding to interpret signals transmitted over the CAN bus [12]. This involves the interpretation of signal levels captured from the CAN transceiver and timing intervals, where each transition—either high to low or low to high is crucial for determining the data bits. These transitions are captured by an Interrupt Service Routine (ISR) configured to trigger on both edges of the NRZ-encoded signals.

B. Real-Time Processing and Rule Enforcement

Upon each ISR activation, the system measures the elapsed time since the last interrupt, using this information to infer the width of the pulse and the corresponding bit value. Instead of assembling complete CAN frames before rule checking, TruckSentry processes incoming data in chunks as it is received. Each data chunk is immediately evaluated against the set of rules designed to identify security breaches.

C. Identification and Impact of the Race Condition

The race condition in TruckSentry is primarily attributed to the concurrent execution of the Interrupt Service Routine (ISR) and the **`get_bitchunk()`** function within the main processing loop, particularly concerning their interaction with the shared variables **`pending`** and **`processed`**. This issue surfaced during system testing on a Kenworth T270 research vehicle [13], where it was noted that the system failed to eliminate certain malicious messages consistently. During the

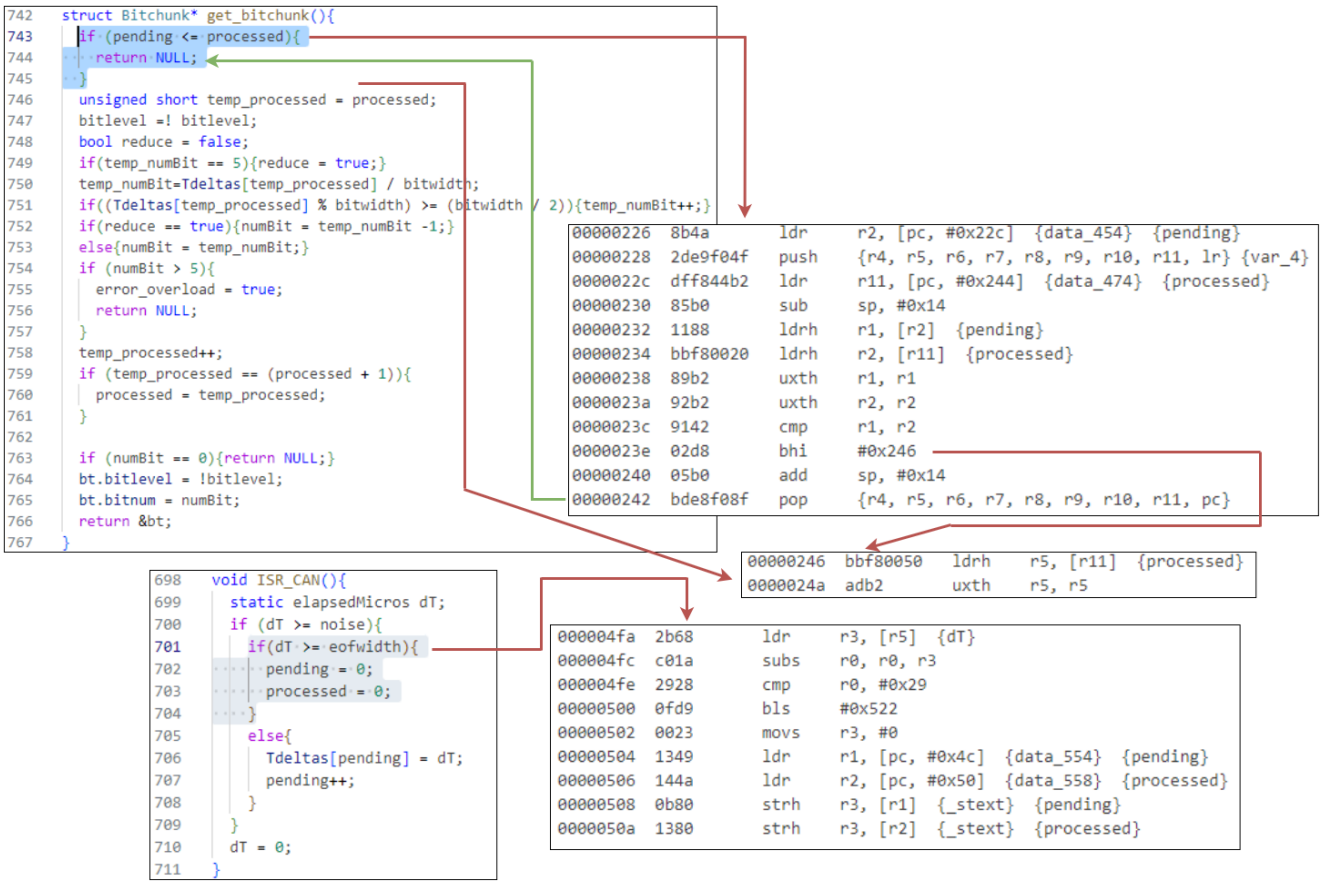

Fig. 2: Source Code that causes race condition along with Decompilation

tests, the CAN identifiers detected by the system were temporarily stored in a buffer and later analyzed. It was observed that some of the CAN identifiers were distorted as shown in Fig.1, deviating significantly from typical patterns expected in an SAE J1939 network. The actual identifiers it should have detected are also shown as a legend in the diagram, by comparing the observations from a logic analyzer. This prompted a deeper investigation into the system's code and behavior under race conditions. A safeguard in the **`get_bitchunk()`** function, designed to halt processing when **`pending <= processed`**, was intended to prevent erroneous data processing. However, this measure was insufficient in addressing the deeper synchronization issue.

To understand the underlying problem, the code was decompiled, and a comparative analysis with the source code was undertaken. Fig.2 illustrates the decompiled view of the ISR, **`ISR_CAN()`** and the **`get_bitchunk()`** function along with parts of the decompiled view. The critical sections highlighted are the checks in both the ISR and the function where if the elapsed time exceeds **`eofwidth`**, the expected end-of-frame duration, both **`pending`** and **`processed`** are reset to 0. In the scenario, an interrupt occurs immediately after **`pending`** is loaded but before **`processed`** is loaded, the ISR might reset these variables. When execution returns to **`get_bitchunk()`**, the function operates with an updated value of **`processed`** but a stale value of **`pending`**. This discrepancy can lead to a critical error where, if pending is at some intermediate state **'x'** and processed is reset to **'0'**, the function erroneously starts to derive bit chunks instead of returning **`NULL`**. This results in an incorrect interpretation of the CAN frame, leading to the system incorrectly interpreting network data. The non-atomic nature of these operations exacerbates the race condition, undermining the system's ability to reliably enforce security rules based on accurate data interpretation.

IV. Solution Implementation

A. CodeHawk Binary Patcher

Developed under the DARPA Assured MircoPatching (AMP) program, the CodeHawk Binary Patcher (CBP) is an end-to-end binary patching platform and associated workflow that significantly reduces the costs of patching stripped binaries without requiring the original source nor a recompilation. This cost reduction is achieved by lifting the binary into a form that can be edited by a typical software engineer (as opposed to the rare and highly-skilled reverse engineer). Concomitantly, our platform drastically decreases the likelihood of errors by formally validating translation steps of the process and by providing the developer with assurance artifacts for review. Furthermore, the platform enacts minimally-invasive changes

Fig. 3: Modifications to lifting in the CodeHawk Patcher

to the binary, with the ability to reason about individual expressions and instructions, maintaining the testing and certifications applied to the original binary.

The Binary Patcher is built on top of the CodeHawk analysis platform [14], [15]. CodeHawk is an industrial-strength binary analysis and patching framework and proven over multiple IARPA and DARPA programs (STONESOUP, AMP, HACCS, STAC, and MUSE). The CodeHawk analysis is based on the mathematical theory of abstract interpretation [16], [17] providing a well-grounded theory for demonstrating correctness of code analysis and verifying translations and modifications of code.

In the binary patching workflow, a developer modifies a high-level C lifting of the function(s) required to fix the flaw. The high-level lifting includes idiomatic source symbol and type usage added to the binary manually or by other tools. After the developer has modified the C lifting to specify the desired changes, CBP will automatically modify the binary to produce the associated semantic changes specified by modifications to the lifting.

Next, the analyst consults automatically-produced assurance artifacts to decide (1) whether the patch was correctly applied to the binary (CodeHawk Patching Validation and Binary Relational Analysis), and (2) whether the patch fixes the flaw (for undefined C behaviors such as memory vulnerabilities and overflows). The translation and analysis steps in the process will produce checkable proofs that can be used to validate each individual step. These checkable proofs provide strong evidence that the step in question is correct.

B. Solution Overview

Fig. 3 provides a screenshot of the patch as modifications to the produced lifting. To quash the ISR race, we wrap the loads of `pending` and `processed` with the assembly instructions `cpsid/cpsie` to disable and then re-enable interrupts (lines 48 and 51, respectively). This ensures that the undesirable interleaving is ruled out. And, thanks to the optimizations detailed below, interrupts are only disabled for a span of two instructions.

C. Binary Analysis Assurance

A stripped binary is a difficult artifact for analysis. The compiler produces a binary with the goal of efficiency and not analyzability. In order to correctly produce our lifting, automated binary modifications, and assurance output, the analysis must understand (and over-approximate) relevant dynamic behaviors of a function. CBP uses its abstract-interpretation engine to generate invariants in multiple domains, which provide the basis for the variable discovery and dataflow relationships necessary to produce a faithful lifting to parseable C code, as well as detailed provenance information that connects the C code with the assembly instructions contributing to each statement. The UI provides a summary of the underlying resolution of these complex binary analysis results. With respect to our solution, both the original and the patched function have 100% of the following resolved (either precisely computed or over-approximated): stackpointer value at each instruction, indirect jumps / calls, memory reads and writes, and conditional (control flow) expressions.

CBP encapsulates its analysis results in a retargetable format called the Patch Intermediate Representation (PIR) [18]. The PIR supplies the patching component of CBP (described next) with the information required to apply complex patches to the binary with minimal changes.

D. Binary Patch Application

The CodeHawk Binary Patcher works by identifying differences (in abstract syntax) of original and user-modified lifted C code. The differences are mapped to corresponding locations in the binary, using metadata embedded in the PIR. Edits to expressions are performed using in-place modifications when possible, while insertions of new statements must be implemented using trampolines.

A trampoline provides the low-level support code necessary to support inserted statements. A trampoline consists of hook, wrapper, and body. The body is a compiled C function containing the user's new code. The wrapper is a snippet of customized assembly code which calls the body and enacts any required control flow from user-written `return`, `break`, or `continue` statements. The hook is an unconditional jump to the wrapper. Any instructions overwritten by the hook are executed by the wrapper.

In the general case the trampoline wrapper must preserve caller-saved registers to avoid accidental corruption of execution state after the trampoline runs. However, in this instance, a more optimized wrapper is possible because the added code does not modify any registers nor does it specify any control flow. The overall structure of the trampolines required for this patch can also be improved by noting that, while the inserted `asm` instructions are separated by two statements at the source level, the corresponding spots in the binary are only four bytes apart. This allows the patcher to emit a single combined trampoline instead of two separate trampolines. The result of these optimizations is that the binary level implementation of this patch is absolutely minimal: a single basic block which simply executes the two loads with interrupts disabled, then

jumps back to continue executing the patched function. The overall patch is a mere 16 bytes: four bytes for the hook, four for the return jump, and eight bytes for the two loads with interrupts disabled.

E. Translation and Patching Validation

CBP seeks to provide strong assurance that it has (1) correctly lifted the semantics of the binary, and (2) correctly applied the changed semantics expressed in the lifing modifications to the binary. For (1), CBP produces a list of the invariants calculated at each instruction of the binary. Invariants are used to produce the lifting. The list of invariants can act as a checkable proof [19] of the invariant calculation (i.e., invariants are expensive to produce but easy to check).

For (2), we are working on a series of validations of the translation steps motivated by prior work on *translation validation* [20]. For our patch, we demonstrate that the (intra-procedural) invariants calculated by CodeHawk's C analysis for the modified lifting exactly match the invariants of the lifted patched function, (in fact, modulo identifier names, the text of the two functions are equivalent in this case). CBP achieves equivalence, in this case, by automatically performing validations on the trampoline, and then essentially inlining the payload of the trampoline in the lifting of the patched function. A broader discussion of our validations is beyond the scope of this paper, but it suffices to say that this particular validation is strong evidence that CBP correctly applied the changed semantics of the modified lifting to the binary.

F. Relational Analysis

To understand the differences that the patch introduces, relational analysis is performed from a few different perspectives: (1) a top-down syntactic/semantic comparison of the entire executable, (2) a visual comparison of the structure of the patch in terms of control flow, and (3) comparison of the invariants generated for the original and patched binary. A brief outline of these perspectives follows.

a) Top-down syntactic/semantic analysis: The minimality of the patch generated by our framework enables a very efficient top-down comparison of the original and patched executable. Taking advantage of the minimality of the patch generated by our platform, much of the comparison can be done syntactically: a simple hash comparison of all functions, and all basic blocks not included in the patch reduces the need for the direct semantic analysis to the single basic block that was modified by the patch. This block is spliced by the patch, inserting a trampoline that surrounds the offending load instructions by an interrupt enable and interrupt disable. Conveniently the two load instructions provide the space for the jump to the trampoline, and thus no other instructions than the two load instructions need to be replicated in the patch, hence the only semantic validation to be performed for the correctness of this patch is that the load instructions are position independent (since they were moved) and indeed load the same value as before. The invariants generated by CBP provide direct evidence for this fact.

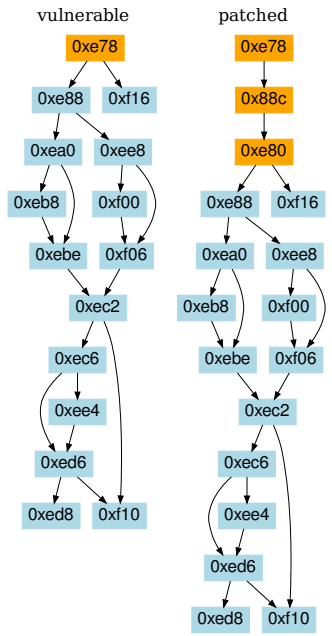

Fig. 4: Visual comparison of the original and patched function

b) Visual comparison: Figure 4 shows the control flow graphs of the original (vulnerable) and patched function. The basic blocks shown in blue are hash-equal in the original and patched function. The basic block at address 0xe78 is modified by inserting the trampoline at 0x88c in the middle. User feedback indicated that figures like these are considered helpful as a quick confirmation that the changes are indeed minimal and have the structure expected.

c) Invariant comparison: As stated earlier, the CBP abstract interpretation engine is used to generate invariants in multiple domains, for both the original and the patched executable, for each instruction address in the target function. A direct location-by-location comparison of all invariants of the entire function (not just the basic block modified), provides a powerful semantic comparison of the effect of the changes in the set of behaviors of the original and patched function. For the patch described here out of the 526 invariants generated for this function the patched function retains 523 of these, losing only the three invariants associated with the instruction overwritten by the jump to the trampoline. This correspondence provides strong evidence that, the patch introduced no semantic changes beyond the interrupt-enable and disable, as desired.

G. Testing and Re-certification

After implementing the patch to address the identified race condition, the updated binary was re-tested on the TruckSentry prototype system. Tests were designed to verify both the resolution of the race condition and the preservation of the system's core functionality as an IDPS. We observed that the device met the performance criteria in real-world testing,

confirming that the patch effectively resolved the issue without compromising the system's functionality. Following these successful outcomes, we proceeded to re-certify the prototype.

V. Discussion

a) Direct Binary Modifications: An astute reader may realize that simply swapping the order of the loads of `pending` and `processed`, i.e., loading `processed` before `pending`, will fix this particular race condition. This change can be enacted by simply swapping the instructions in the Binary Ninja assembly editor. However, the CodeHawk relational patch analysis demonstrated this did not work via a manual review of the changed invariants. To summarize, the first instruction uses a temporary in a register that is overwritten by the second load. The next attempt was to instead rewrite the preceding instructions which performed PC-relative loads of global addresses into temporary registers. By changing which address was held in which temporary, the order of the loaded values could be swapped without needing to modify the target load instructions themselves. However, manual review of the function invariants revealed that one of the temporaries was reused on a later path. In both cases, using CodeHawk to analyze the resulting patched binaries yielded invariants which clearly flagged the unintended semantic alterations. Having failed to produce a correct patch by manual assembly modifications, utilizing Codehawk Binary Patcher provided a simpler workflow for creating a more complex patch.

b) Reduction in Required Experiences for Binary Patching: Currently, binary patching is a costly process due to the skills and experience required. It is also error prone for a multitude of reasons including a lack of robust tools for reasoning about differences between binaries. CBP reduces the cost of patching legacy systems, since in many situations it does not require manual investigation nor understanding of the underlying binary instructions. A reverse engineer is not required; instead, an experienced C developer can patch a stripped binary. Furthermore, CBP produces assurance results understandable by a developer, not a specialist in program analysis. The assurance results are provided as validation checks, lifting differences, C code invariants, and C behavior checks. The patching workflow has been successfully employed by independent evaluation teams on DARPA's Assured Micropatching program to create and assure binary patches.

VI. Conclusion

Interrupt service routines triggered by external events through hardware may induce race conditions. The symptoms of these issues are stochastic, which can make detection during development challenging. Once discovered mitigating the issue requires a deep understanding of the binary executable, a patch to change or fix the issue, and an assurance analysis to show non-interference of the patch.

Our study using the CodeHawk Binary Analysis system within an IDPS for commercial vehicle networks illustrates its efficacy in mitigating such issues, even without source code access. This application underscores CodeHawk's broader potential for binary-level modifications across various systems, demonstrating its capacity to swiftly enhance security and maintain system integrity against emerging threats.

References

[1] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[2] M. Jones, "What really happened on Mars?" *The Risks Digest*, vol. 19, no. 49, 1997, available online: http://catless.ncl.ac.uk/Risks/19.49.html.

[3] A. Muir and J. Lopatto, "Final report on the August 14, 2003 blackout in the United States and Canada: causes and recommendations," 2004.

[4] R. Chen, X. Guo, Y. Duan, B. Gu, and M. Yang, "Static data race detection for interrupt-driven embedded software," in *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement - Companion*, 2011, pp. 47–52.

[5] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 237–252. [Online]. Available: https://doi.org/10.1145/945445.945468

[6] Y. Wang, F. Gao, L. Wang, T. Yu, J. Zhao, and X. Li, "Automatic detection, validation, and repair of race conditions in interrupt-driven embedded software," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 346–363, 2022.

[7] S. Mukherjee, "SAE J1939-specific cyber security for medium and heavy-duty vehicles," Ph.D. dissertation, Colorado State University, Fort Collins, Colorado, 2023. [Online]. Available: https://mountainscholar.org/items/95d89caa-b783-439f-b763-f89ec0c0b7e2

[8] "CodeHawk GitHub Repo," https://github.com/static-analysis-engineering/codehawk.

[9] R. H. B. Netzer and B. P. Miller, "What are race conditions? some issues and formalizations," vol. 1, no. 1, p. 74–88, mar 1992. [Online]. Available: https://doi.org/10.1145/130616.130623

[10] G. Karam, C. Stanczyk, and G. Bond, "Critical races in Ada programs," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1471–1480, 1989.

[11] T. R. Allen and D. A. Padua, "Debugging Fortran on a shared memory machine," 1987. [Online]. Available: https://api.semanticscholar.org/CorpusID:59905983

[12] Robert Bosch GmbH, "CAN Specification," Robert Bosch GmbH, Standard 2.0, 1991.

[13] J. Daily, "The CyberTruck Research Vehicle." [Online]. Available: https://www.engr.colostate.edu/~jdaily/truck/index.html

[14] H. B. Sipma, "A Gold Standard for Benchmarking C source code static analyis tools: Measures and Metrics," Kestrel Technology, LLC, Tech. Rep., May 2013.

[15] ——, "Sound Static Analysis of C Programs and x86 Executables using Abstract Interpretation," Kestrel Technology, LLC, Final Report, DHS-sponsored, AFRL-contract: FA8750-12-C-0277, 2016.

[16] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 1977, pp. 238–252. [Online]. Available: https://doi.org/10.1145/512950.512973

[17] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, A. V. Aho, S. N. Zilles, and T. G. Szymanski, Eds. ACM Press, 1978, pp. 84–96. [Online]. Available: https://doi.org/10.1145/512760.512770

[18] "Patching Intermediate Representation," https://github.com/static-analysis-engineering/CodeHawk-Binary/tree/master/chb/ast/doc.

[19] G. C. Necula and P. Lee, "Safe, untrusted agents using proof-carrying code," in *Mobile Code Workshops*. Springer, 1996.

[20] M. C. Rinard and D. Marinov, "Credible compilation with pointers," in *Proceedings of the FLoC Workshop on Run-Time Result Verification*. ACM, 1999.

View publication stats