RIPEMB: A framework for assessing hardware-assisted software security schemes in embedded systems

Stefan Tauner
stauner@ecs.tuwien.ac.at
TU Wien
Embedded Computing Systems Group (ECS)
Wien, Austria

ABSTRACT

Memory corruption bugs remain one of the biggest threats to software security. The increasing complexity of SoCs and prevalence of connected embedded devices require larger software support packages that inevitably contain more bugs. Unfortunately, as of now, hardware-assisted security measures are not widely available in smaller embedded devices based on MCUs. Even if they are, vendors might configure them inadequately and validating the correct behavior of such important features is advisable.

In this paper, we present RIPEMB, an open-source software package for validating hardware-assisted protection mechanisms such as memory protection units (MPUs), control flow integrity (CFI) enforcement, code pointer integrity (CPI), data flow tracking etc. It works as a self-contained embedded application performing up to almost 3000 different attacks based on memory corruption. While it contains some target-specific components, it is easy to port to new environments and can be used during development of new security schemes and in validation alike. We evaluate the applicability on two instruction set architectures (ISAs) (ARM and RISC-V), four hardware platforms, two C runtime environments, and a total of 8 different hardware defenses.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**.

ACM Reference Format:
Stefan Tauner. 2022. RIPEMB: A framework for assessing hardware-assisted software security schemes in embedded systems. In *The 17th International Conference on Availability, Reliability and Security (ARES 2022), August 23–26, 2022, Vienna, Austria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3538969.3539013

1 INTRODUCTION

In the last years, security research in academia has increased massively in some fields like microarchitectural side channels (e.g., SPECTRE [13]) or vulnerabilities and countermeasures to attacks in deep neural networks (DNNs) [18]. However, in practice more mundane problems continue to haunt administrators and software developers alike: Traditional security vulnerability categories like *out-of-bounds buffer access* and *use-after-free* continue to have the biggest impact in the wild [2].

Prominent examples are the notorious class of stack buffer overflows [16] and format string vulnerabilities [20]. These can be used to divert the control flow by corrupting data in memory, namely code pointers, and disclose secrets like passwords, cryptographic keys, or defeat address space layout randomization (ASLR) [21].

Academia has presented a plethora of approaches to mitigate these problems from safer new programming languages, over probabilistic software mechanisms, to extended microarchitectures. In many cases the evaluations of these works focus on benchmark programs to show the overheads in execution time, code size, memory usage etc. and to prove the absence of obvious regressions, i.e., that the new security features do not perturb the regular operation. Determining overheads of security provisions comes with its own set of difficulties [14] but arguably even more important is demonstrating the effectiveness of averting attacks in a sound way. Nevertheless – albeit being the main motivation behind their contributions – security is often merely discussed without systemically evaluating the implementation in practical terms that go beyond trivial test cases.

Also, security research often focuses on Intel's x86 architecture, which has led to some blind spots in regards to embedded systems. However, the connectivity of ever wider spreading IoT devices in this domain requires additional defenses that could largely be ignored in the past without dramatic consequences. Simultaneously, the complexity of these devices' hardware as well as software has increased significantly. Therefore, vendors have to provide large SDKs or even code generators that often rely on third-party libraries (e.g., real-time operating systems (RTOSes) implementations like FreeRTOS). Unfortunately, many hardware vendors are notorious for the lack of quality in their software packages. Over the years we have discovered numerous problems in the libraries provided by different vendors that clearly show a complete lack of awareness for the importance of quality assurance. Other engineers with five-decade-long experience describe these circumstances in less polite words [19]. Therefore, we argue that any hardware provisions for security in these systems require additional scrutiny due to their importance in helping to secure the IoT. System integrators would also benefit from being able to independently verify hardware-assisted software security schemes.

In this paper, we present RIPEMB, a cross-platform software package for evaluating hardware-assisted software security schemes in embedded systems. Compared to its predecessor, Runtime Intrusion Prevention Evaluator (RIPE) [26], it has been improved in

ARES 2022, August 23–26, 2022, Vienna, Austria

ACM ISBN 978-1-4503-9670-7/22/08.
https://doi.org/10.1145/3538969.3539013

several ways, for example, it (1) does neither require operating system services nor an external execution manager, (2) relies on vastly less implementation-defined and undefined behavior allowing for easier reuse, (3) provides facilities to automatically set up and reset security mechanisms between test cases, (4) comprises a refined set of attacks better suited for freestanding runtime environments.

In the next section we discuss the related work and the shortcomings of RIPE for our use case. In the Implementation chapter we explain our approach in detail and in Section 4 we document the evaluation in different environments. Finally, we present an outlook on further research and our conclusion.

2 RELATED WORK

There is no shortage of buggy source code in the world [no citation needed]. A tremendous amount of work has been put into contriving categorization systems to classify and group software defects, e.g., Common Weakness Enumeration (CWE). These allow researchers and practitioners to systematically dissect problems in real-world projects and measure their distribution and severity, e.g., in combination with the Common Vulnerability Scoring System (CVSS).

Furthermore, various groups have been collecting programs and code snippets containing known flaws [7, 15, 22]. They are created by injecting artificial vulnerabilities into production software, extracted from reported actual vulnerabilities (e.g., the Common Vulnerabilities and Exposures (CVE) system), or generating synthetic snippets (manually or automatically). These can be used as examples for educating developers but their main purpose is to function as test cases for tools providing static analysis, vulnerability detection, penetration testing, automatic attack and patch generation etc. When comparing respective tools by using these code collections, it is important to validate that they are representative for the intended environment since overall they don't correlate with the distribution of real-world weaknesses [3].

For the purpose of our paper the existing libraries are of little help unfortunately. While there are plenty of examples written in C, most of them are not suitable for freestanding environments (i.e., without an operating system (OS)). Furthermore, creating a weakness is only the first step that is only sufficient for the main use case of these collections, where the code is never run but just analyzed.

However, to realistically mimic an attack as we require, the flaw has to be exploited to modify the state of the application, e.g., by alternating the control flow and/or accessing memory malevolently. In our application, the execution flow must be kept under control to some extent since we cannot rely on an OS to clean up after every test case. This demands rigor when constructing the attacks, which reduces the available options to procure suitable programs significantly and only leaves those instances that include a proper exploit including the correct inputs to not only trigger the vulnerability but remain in control. Unfortunately, these are almost exclusively targeting hosted environments (i.e., applications running on top of an OS), which we don't have at our disposal.

2.1 RIPE's History

While not a vulnerability collection in the traditional sense and although it is bound to an OS, the RIPE project serves as the base for our implementation [26]. Its principal part is a C program that is able to set up various attacks on itself and execute them. RIPE itself is based on a tool written by John Wilander and Mariam Kamkar for their evaluation of run-time mechanisms that (try to) prevent buffer overflows [25]. The work investigates the various early attempts by run-time libraries and compiler extensions to secure return addresses and function pointers in the early 2000s, e.g., StackGuard [8]. The tool itself tests for 20 possible buffer overflow attacks combining values from three sets of properties (later termed *dimensions* by Wilander et al):

- 2 *techniques*: either directly overwriting the target address or indirectly manipulating a pointer to point to that address and manipulating it through that.
- 4 *locations*: buffers stored on the stack or in the heap, BSS or data section.
- 6 *targets*: function return address, the previous base/frame pointer, function pointers (as local variables or function parameter), and `setjmp`/`longjmp` buffers (local or parameter too).

RIPE extends the previous work "in an attempt to standardize the comparison between countermeasures and to further support research on code-injection countermeasures" [26]. It adds two more dimensions:

- 10 *functions* that are used to overflow the respective buffers, including notorious string functions (e.g., `strcpy`).
- 5 *attack code* modes: three shellcodes with different types of NOP sleds, one return-to-libc and return-oriented programming (ROP) attack, respectively.

In both works the results of protection schemes based on software alone were devastating while they simultaneously showed the effectiveness of simply declaring data memory non-executable and enforcing this by hardware (via no-execute (NX)/execute disable (XD) features of x86-CPUs). The latter thwarts all attacks that introduce new instructions (i.e., shellcode) by writing them to data memory and manipulating the execution flow to jump to them.

RIPE relies on an external execution manager in form of a Python script (cf. Fig. 1 on Page 4). This script launches the main application (the *Attack Generator*) that is written in C with different command line arguments configuring the respective attack. Every instance is only executing a single attack and cleaning up is completely left to the OS. This architecture is very disadvantageous in embedded systems: Starting applications often take a long time as they have to be downloaded from a host PC to the device's flash or RAM. There is no OS that is able to clean up failed attempts or even detect them, nor to keep track of the progress and results. Additionally, some of the attacks require an OS themselves to execute because they use system calls and file streams.

Furthermore, the implementation of the *Attack Generator* has some significant issues as it unnecessarily relies on a very peculiar layout of variables by the compiler that breaks when slightly changing the environment, e.g., upgrading the compiler or changing its configuration. These makes it hardly possible to use RIPE

reliably in different environments and complicates porting it to other platforms.

Although the source code of RIPE has been publicly available for over 10 years and the respective paper has been cited over 100 times, very little (public) development has been happening. Two noteworthy exceptions are the port to ARM created for the evaluation of FastCFI [9] and to RISC-V [11] as they target non-x86 architectures. The former modification is rather crude comprising only the minimal changes needed to make RIPE run in a Linux environment on ARM. It contains hard-coded addresses for the frame layout used to attack function returns.

The RISC-V port is targeting the 64-bit version of the Spike instruction set simulator (ISS) [4]. It does not alleviate the main problems of the original version but provides two notable improvements: (1) The constant shellcode is replaced by instructions dynamically generated at runtime that call a given address. (2) Some data-oriented attacks have been added, e.g., an information leak and a restriction bypass.

In the next section we describe how the mentioned problems can be solved and how our modifications can improve RIPE's application in embedded systems.

3 IMPLEMENTATION

Our aim is to allow for validating hardware-assisted security provisions in embedded systems without an OS. To refine RIPE for this purpose, we first analyze its architecture.

3.1 RIPE's Legacy

The original RIPE architecture (depicted in Fig. 1) splits the management of the test execution between three parts:

RIPE Tester. The main user interface of RIPE is a Python script that essentially iterates through all possible combinations of dimension values to set up the list of arguments to be passed to the *Attack Generator* via Python's `os.system()`, which is a simple wrapper around C's `system()` function. It is also responsible for managing the two files used to communicate with the *Attack Generator* and evaluating the result of every run. Furthermore, it aggregates and prints the final results.

Operating System. RIPE exploits the operating system's capabilities to handle processes that together with the memory management unit (MMU) are able to detect and stop the *Attack Generator*, if it accesses memory addresses outside the process' legal address range or if it produces exceptions, e.g., by executing illegal instructions. However, in the current implementation RIPE would wait forever if the *Attack Generator* ends up in an infinite loop. Also, the reason for the end of the *Attack Generator* processes is not taken into account by the Python script. No matter why the result file (i.e., `/tmp/rip-eval/f_xxxx`) is not created by a run of the *Attack Generator*, it is counted as a failed attack. This might happen either because of a security mechanism thwarting the attack or due to a problem within the *Attack Generator* (e.g., by deliberately exiting because a buffer is misaligned or due to some peculiarity of the attack). Conversely, it does not matter if the process is killed after it created the file.

Attack Generator. The actual attacks are contained within an ordinary C application that comprises the necessary flaws as well as the necessary code to exploit them. The program's parameters specify the values of the five dimensions that eventually determine which attack is to be executed. The *Attack Generator* checks if the values make any sense and signals impossible combinations back to the *RIPE Tester* via a log file. If the attack is valid, the *Attack Generator* continues by setting up numerous buffers, initializing the target address to overwrite, before corrupting the data via some deliberately included weakness (e.g., by copying a prepared payload into a too small buffer). Finally, the exploit is triggered by executing an innocent statement that relies on the previously manipulated data (e.g., returning from a function or de-referencing a function pointer).

3.2 RIPEMB

Since we do not have a full-grown OS nor processes at our disposal on microcontrollers, we have to refine all of the respective functionalities and provide alternatives.

To replace the Python script, we move the nested loops to generate all possible attack combinations into the *Attack Generator*. To aid development and debugging, we keep the possibility of executing individual attacks alone via compile-time options. Furthermore, it is possible to lock individual dimensions to a single value and iterate over the remaining ones.

The accounting of results has been integrated into the *Attack Generator* and extended to distinguish between several possible outcomes. RIPEMB counts the following result types individually in addition to the statically impossible one of original RIPE:

`Dynamically Impossible` attacks fail a run-time check, e.g., if a payload contains null characters while the attack vector is a string function (as the payload would be cut short).

`Setup Error` logs unexpected problems before an attack.

`Failed` is counted in cases where the attacks fail without any observable ill effects (i.e., if the control flow continues on the same path as it would do without corruption).

`Detected` allows implementations to count successfully detected attacks.

`Illegal Instruction` can be used by implementations to count recoveries from botched attacks.

`Successful` is only counted if the program follows the intended attack path, e.g., not only jumping into shellcode but also successfully calling the target function.

If RIPEMB works as intended, only `Detected` and `Successful` are logged for attacks that are statically and dynamically possible. Note that distinguishing between `Failed` and `Detected` is fundamental to validate security mechanisms as a failed attack might not necessarily be rooted in a successful defense. The other possible outcomes (`Setup Error`, `Failed`, and `Illegal Instruction`) should always remain zero if a platform including any respective security mechanisms is properly supported.

3.2.1 Retaining Control. The biggest challenge in consecutively running hundreds of exploits in a freestanding environment is to not crash completely nor trash essential data structures to separate cause-effect relations among different attacks.

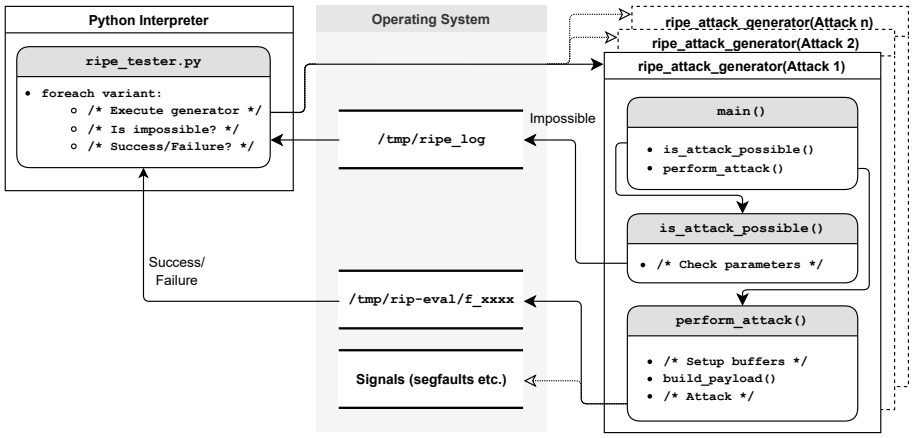

Figure 1: Overview of the original RIPE architecture.

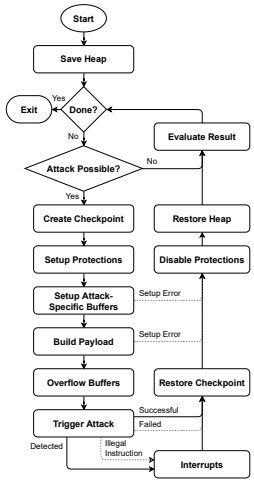

Figure 2: Flow diagram of RIPEMB.

Figure 2 shows the execution flow of a full run of RIPEMB. At the very beginning RIPEMB mirrors the current heap memory including management data into a backup location that is restored at the very end of every round. That way, even corruption of the heap metadata is not fatal in future iterations. However, it requires a save/restore implementation that is tailored to the used heap manager, which is usually bundled with the C runtime library. All other memory sections do not need such a safeguard as RIPEMB's attacks on variables within them are precise enough to not tamper with surround data, which cannot be guaranteed for the heap due to its implementation-defined metadata that is often intertwined with user data.

Afterwards, the nested loops iterate through all (enabled) combination of dimension values, which are checked for validity. The execution of each attack iteration heavily depends on the use of `setjmp`/`longjmp` to retain control and jump back to a defined location. To that end, we use `setjmp` at the beginning to create a checkpoint that can then be used in all kinds of situations, e.g., if an attack succeeds that ends up in a function called by shellcode.

The next step allows for configuring any security mechanisms, e.g., enable hardware units, set attributes for memory ranges etc. This and its inverse that is executed before restoring the heap are optional.

Then the various memories used in the respective vulnerability have to be set up according to the current attack. As many of the attacks rely on overflows, the actual layout and relative positions of the used buffers are critical. The C language does not specify the relationships between unrelated variables and the compiler is free to rearrange them as it sees fit if no constraints are given. This made the previous implementation very unreliable as it depended on a peculiar layout of local, static and global variables that could change with compiler updates, changed optimization settings etc. Fortunately, the C standard at least guarantees the order of objects in one type of object: aggregates (i.e., structs and arrays): "If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure" [12, § 6.5.8.5]. The compiler is allowed to add some padding between structs members but this is benign if taken care of by calculating the correct offsets (e.g., to include them in the data used to overflow a buffer). For that reason, RIPEMB keeps all variables that require some sort of relative order to each other within structs.

Finally, the payload containing the address value to be overwritten and potentially some shellcode is created and used to overflow the respective buffer. The attack is then triggered by de-referencing a function pointer, restoring a return address etc. that has been manipulated by the overflow. Unlike RIPE, the new implementation then uses `longjmp` to return from there to the previous checkpoint passing information about the status.

To facilitate this level of control, platform-specific ports need to implement some additional support functions. For example, any security measure must pass `RET_ATTACK_DETECTED` to a dedicated `longjmp` function (that does not trigger the security reaction recursively) to inform the implementation about any unexpected deviation from the intended control flow or memory access patterns it detects.

3.2.2 Additional Improvements and Limitations. Apart from the unified and stable execution, we refined some aspects of RIPE's attacks as well.

The three different shellcodes have been replaced by a single (architecture-dependent) one that is generated dynamically at runtime. The return-to-libc attack is simulated by targeting an ordinary dedicated custom function to make it applicable in freestanding environments (without `system()` and the like). No (fake) parameters have to be passed to this function, making it universally applicable even on platforms with application binary interfaces (ABIs) where these parameters might not be reachable on the stack because they are passed in registers. Similarly, `fscanf()` has been removed as attack vector and `sscanf()` can be turned off at build time in case it is not available in the respective C library.

The target variables have been cleaned up as explained in the previous section and now comprise function pointers and longjmp buffers in parameters as well as in all four memory sections. We

	STM32F	STM32L	TM4C	PULPissimo
ISA	Armv7-M	Armv7-M	Armv7-M	RISC-V
Core	C.-M4F	C.-M4F	C.-M4F	CV32E40P
F_{max} [MHz]	168	80	120	16
Perf. [DMIPS]	210	100	150	11
RAM [KiB]	192	64	256	512
Flash [KiB]	1024	256	1024	–
C Runtime	Newlib	Newlib	Newlib	PULP
GCC Version	7.3.1	7.3.1	7.3.1	7.1.1

Table 1: Evaluation platforms.

have adopted two of the data-oriented attacks by John Merrill as they strikingly depict some limitations of defensive mechanisms against control-oriented attacks. Furthermore, we have added more fine-grained variants of the control-oriented attacks to further distinguish techniques forCFI enforcement: For each attack using the return-to-libc and shellcode principles, a variant targeting a function that is legally called indirectly (i.e., that is contained in the intended control flow graph (CFG)) has been added. Additionally, we target an ancestor function, i.e., one having a respective frame in the stack trace, with an ROP and return-to-libc attacks, respectively.

4 EVALUATION

4.1 Platforms

We have evaluated RIPEMB on four different hardware platforms with 32-bit CPUs as summarized in Table 1. Three are commercial MCUs from STMicroelectronics and Texas Instruments with single-core Cortex-M4 CPUs by ARM mounted on typical development boards. The fourth one is an open-source SoC based on the CV32E40P (formerly RI5CY) CPU core originally developed by the PULP project [10]. For our evaluation we synthesized it to run in a Xilinx Zynq Z-7020 FPGA with a CPU frequency of 16 MHz.

4.2 Security Mechanisms

In our evaluation we employ various different CFI enforcement schemes that extend PULPissimo. Additionally, a memory access restriction system already present on the ARM platforms is tested.

4.2.1 Label-based CFI. The majority of tested defenses use different CFI mechanisms based on labels, i.e., they maintain a connection between caller and callees based on additional instructions added by the compiler similar to what Abadi et al. have proposed in their seminal paper [1]. We use six hardware-assisted implementations that differ in the amount and locations of extra instructions as well as their coverage and granularity [24].

4.2.2 CFI via Pointer Authentication. Another HW-assisted CFI approach we used is based on pointer authentication (PA), which comprises some extra instruction to cryptographically authenticate pointers. We have implemented a scheme somewhat similar to Authenticated Call Stack (ACS) [17], where the return address used in `ret` instructions is authenticated before use to guarantee that it has not been modified. To that end, we have extended PULPissimo to support the necessary instructions and use a hardware implementation of the QARMA block cipher [6]. Due to the smaller addresses of our 32-bit MCU, we use only 13 bits to store the pointer authentication code (PAC) in contrast to ACS's 16.

4.2.3 Memory Protection Unit. The Armv7-M architecture specification [5] defines an optional MPU that limits possible accesses on configurable memory regions, which can be as small as 32 B. It is not a full memory management unit (MMU) and does not perform address translation but grants access to physical addresses. If a program accesses a location that is prohibited by the MPU, the processor generates a memory management fault. Most implementations restrict the number of supported regions fairly low, e.g., to 8 in our devices. Each region can be set to limit read, write, and execute/instruction accesses for privileged/unprivileged software, respectively.

For the evaluation of RIPEMB we have exploited the MPUs of the three ARM MCUs to thwart all attacks that use shellcode by prohibiting instruction fetches from all data memories (e.g., stack).

4.3 Results

While the assessed defenses differ in capabilities, precision, and scope, RIPEMB is able to successfully determine their individual limitations. Currently it executes up to 2803 individual attacks. However, in most practical scenarios the involved addresses contain 0-bytes and thus drastically reduce this number since string functions (e.g., `strcpy`) cannot be used as vulnerable target functions because they would abort early when encountering the ostensible string delimiters during the overflowing call. For example, on the STM32L target only 822 attacks remain of which 110 are based on executing shellcode, which is detected by the MPU. We refrain from assigning the exact numbers of detected attacks to the individual security measures here to not give the false impression that these can be interpreted as some kind of security benchmark, which they clearly aren't. The purpose of RIPEMB is to validate that theoretical predictions about a defense mechanism concur with the practical implementation, not to compare security mechanisms numerically with each other. You can find an elaborate report at the RIPEMB website and are invited to send us results of your own projects.

The runtime is primarily limited by the output method (e.g., UART), the selected verbosity, and the number of possible tests. On the STM32L hardware with a 1 MBaud connection a complete run without MPU protection (i.e., with 822 successful attacks) takes under 30 seconds.

5 CONCLUSION

Fully-fledged computer systems have had to deal with adversarial users for decades and have therefore been continuously hardened against malicious intents. There is no doubt that there are still numerous open issues in the security domain in these environments and even the wider society has recognized this as a problem. The contrary is true for embedded systems: While the attack surface and prevalence of microcontroller units (MCUs) increases steadily, the amount of resources to secure them is scarce and often focuses on protecting intellectual property of vendors instead of the data of end users. Our work aims at providing security researchers and embedded engineers alike an easily adaptable tool to help alleviate this imbalance.

In this paper we have presented RIPEMB, an open-source software package that mounts memory corruption attacks on itself to challenge hardware-assisted security mechanisms in MCUs. It specifically improves on the state of the art by removing the requirement to utilize an operating system to contain the ramification of the attacks. RIPEMB …

- precisely controls the memory layout of the allocated buffers and the execution of the attacks,
- provides facilities to easily encapsulate platform-dependent code,
- keeps a shadow copy of the application's heap memory to avoid corruptions from influencing the individual tests,
- allows for easily integrating the necessary instructions to configure HW security entities.

We have shown that RIPEMB can be applied to different types of defense mechanisms and hardware platforms. It is freely available for download [23].

5.1 Future Work

While RIPEMB uses the heap memory in its attacks, e.g., as one possible location for shellcode, it does not directly attack the heap metadata yet. To avoid impairing platform-independence when adding such attacks, a heap management library independent from the system's own heap manager seems to be the best option. Support for integer over- and underflows are also a viable enhancement goal.

Extending RIPEMB to cooperate with an RTOS could test potential defense features like task separation. Adding further support for running RIPEMB in hosted environments would allow for sharing new attacks and other improvements between systems on the whole spectrum.

ACKNOWLEDGMENTS

This work has been supported by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien. We would also like to thank Andreas Steininger and all other reviewers of the manuscript for their helpful feedback. Additionally, thanks are due to John Wilander and Nick Nikiforakis as well as John Merrill and Arun Thomas for open-sourcing their work on RIPE.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (Oct. 2009). https://doi.org/10.1145/1609956.1609960

[2] Adam Chaudry, Steve Christey Coley, Kerry Crouse, Kevin Davis, Devon Ellis, Parker Garrison, Christina Johns, Luke Malinowski, Rushi Purohit, Becky Powell, David Rothenberg, Alec Summers, and Brian Vohaska. 2021. 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[3] Kayla Afanador and Cynthia Irvine. 2020. Representativeness in the Benchmark for Vulnerability Analysis Tools (B-VAT). In *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association. https://www.usenix.org/conference/cset20/presentation/afanador

[4] Andrew Waterman, Chih-Min Chao, Tim Newsome, and Scott Johnson. 2022. Spike RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim

[5] ARM. 2021. *Arm V7-M Architecture Reference Manual*. Spec DDI 0403E. ARM. https://developer.arm.com/documentation/ddi0403/latest Issue E.e.

[6] Roberto Avanzi. 2017. The QARMA Block Cipher Family. *IACR Transactions on Symmetric Cryptology* (March 2017), 4–44. https://doi.org/10.13154/tosc.v2017.i1.4-44

[7] Paul E. Black. 2017. SARD: Thousands of Reference Programs for Software Assurance. 5, 3 (Oct. 2017), 6–13. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=923127 Last Modified: 2021-05-04T09:23-04:00.

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium*. USENIX Association, San Antonio, TX, USA. https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention

[9] Lang Feng, Jeff Huang, Jiang Hu, and Abhijith Reddy. 2019. FastCFI: Real-Time Control Flow Integrity Using FPGA Without Code Instrumentation. In *Runtime Verification*, Bernd Finkbeiner and Leonardo Mariani (Eds.). Springer, Cham, 221–238. https://doi.org/10.1007/978-3-030-32079-9_13

[10] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. 2017. Near-Threshold RISC-V Core with DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (Oct. 2017), 2700–2713. https://doi.org/10.1109/tvlsi.2017.2654506

[11] John Merrill and Arun Thomas. 2018. Hope-RIPE. The Charles Stark Draper Laboratory, Inc.. https://github.com/draperlaboratory/hope-RIPE

[12] JTC 1/SC 22/WG 14. 2018. *Programming Languages — C*. Std 9899:2018. ISO/IEC. https://www.iso.org/standard/74528.html

[13] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, USA, 19–37. https://doi.org/10.1109/SP.2019.00002

[14] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2019. SoK: Benchmarking Flaws in Systems Security. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 310–325. https://doi.org/10.1109/EuroSP.2019.00031

[15] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (Dec. 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[16] Elias Levy. 1996. Smashing the Stack for Fun and Profit. *Phrack* 7, 49 (Nov. 1996). http://phrack.org/issues/49/14.html

[17] Hans Liljestrand, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. 2019. Authenticated Call Stack. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, Las Vegas, NV, USA. https://doi.org/10.1145/3316781.3322469

[18] Sparsh Mittal, Himanshi Gupta, and Srishti Srivastava. 2021. A Survey on Hardware Security of DNN Models and Accelerators. *Journal of Systems Architecture* 117 (May 2021), 30. https://doi.org/10.1016/j.sysarc.2021.102163

[19] Dave Nadler. 2021. FreeRTOS Helpers. https://github.com/DRNadler/FreeRTOS_helpers

[20] scut. 2001. *Exploiting Format String Vulnerabilities*. Techrep. Team TESO. https://cs155.stanford.edu/papers/formatstring-1.2.pdf

[21] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62. https://doi.org/10.1109/SP.2013.13

[22] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 180–182. https://doi.org/10.1109/ICSE-C.2017.76

[23] Stefan Tauner. 2022. RIPEMB Website. https://ripemb.github.io

[24] Stefan Tauner and Mario Telesklav. 2021. Comparative Analysis and Enhancement of CFG-Based Hardware-Assisted CFI Schemes. *ACM Transactions on Embedded Computing Systems* 20, 5s, Article 58 (Sept. 2021). https://doi.org/10.1145/3476989

[25] John Wilander and Mariam Kamkar. 2003. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *NDSS Symposium 2003*. Internet Society. https://www.ndss-symposium.org/ndss2003/comparison-publicly-available-tools-dynamic-buffer-overflow-prevention/

[26] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime Intrusion Prevention Evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/2076732.2076739