Exploring the Environmental Benefits of In-Process Isolation for Software Resilience

Merve Gülmez
Ericsson Security Research
Kista, Sweden
imec-DistriNet, KU Leuven
Leuven, Belgium
merve.gulmez@kuleuven.be

Thomas Nyman
Ericsson Product Security
Jorvas, Finland
thomas.nyman
@ericsson.com

Christoph Baumann
Ericsson Security Research
Kista, Sweden
christoph.baumann
@ericsson.com

Jan Tobias Mühlberg
imec-DistriNet, KU Leuven
Leuven, Belgium
Université Libre de Bruxelles
Brussels, Belgium
jan.tobias.muehlberg@ulb.be

***Abstract*—Memory-related errors remain an important cause of software vulnerabilities. While mitigation techniques such as using memory-safe languages are promising solutions, these do not address software resilience and availability. In this paper, we propose a solution to build resilience against memory attacks into software, which contributes to environmental sustainability and security.**

I. Introduction

The urgent need for climate action requires a focus on sustainability in every field, including security [5]. Recent work attempts to define numerous different notions for sustainable security [12], [15]. For example, Paverd et al. [15] define design principles for sustainable security and safety, such as isolation between components, replication, and diversification. However, these principles relate to sustaining the security of the system rather than environmental sustainability. In this paper, we discuss how building resilience into software against memory attacks can, by limiting the need for redundancy in dependable systems software, contribute to environmental sustainability in addition to sustaining security.

Memory-unsafe programming languages such as C, and C++ are still one of the primary root causes of software vulnerabilities [3]. These vulnerabilities can allow attackers to access vulnerable systems by compromising program behavior. There are two ways to address memory-related attacks: 1) retrofit the unsafe software with run-time defense techniques, 2) perform new development in memory-safe languages.

While well-known mitigations such as control-flow integrity and stack canaries can detect attacks, they stop the attacks by terminating the application. Service-oriented applications are particularly at risk because a temporary failure can have an impact on a large number of clients. Consequently, designing systems for resilience additionally requires replication or redundancy. Such over-provisioning, however, is not environmentally friendly. There is a need to address software resilience through approaches that are environmentally sustainable and also built upon principles of sustainable security.

Rust is an emerging memory-safe language with performance nearly as good as conventional system programming languages such as C and C++ [16]. Rust is more difficult to exploit by design, as it forbids unsafe memory access patterns and enforces a strong type system. However, Rust provides foreign function interfaces (FFI) that enable an application to use system libraries written in other languages, essentially calling memory-unsafe code. This can violate the memory safety of Rust code, e.g., by dereferencing raw pointers or breaking ownership rules [11]. Even if such applications are mainly written in Rust, there is no guarantee that the application is resilient against memory corruption attacks. Efficient approaches for software resilience could benefit Rust applications that rely on unsafe code through FFI as well.

In earlier research we introduce *Secure Rewind and Discard of Isolated Domains* [6], as briefly described in § II. Our approach leverages mechanisms for hardware-assisted in-process isolation in commercial, off-the-shelf processors and allows retrofitting C applications with the capability to recover from memory errors in isolated components. However, a drawback of this approach is compartmentalizing an application into distinct, isolated domains as a pre-requisite for secure rewind can require code changes and manual effort by developers. That drives up the cost of software development, both in terms of money and energy consumption.

To address this drawback, and to extend the approach to protect FFI functionality, preserving the memory-safety guarantees of Rust applications, we are currently adapting our secure rewind library to a streamlined version that allows developers to leverage metaprogramming in Rust to annotate functions that must be compartmentalized, as discussed in §III.

In future work, we plan to quantify the benefit of secure rewind in terms of benefits to environmental sustainability gained through improved resilience against attacks. We discuss possible methodologies in § IV.

II. Secure Domain Rewind and Discard

Our *Secure Rewind and Discard of Isolated Domains* scheme [6] allows restoring the execution state of a program to a state in which allocated memory is free from corruption. This is possible with two pre-requirements: 1) Compartmentalizing an application into distinct domains by leveraging hardware-assisted software fault isolation which guarantees that a memory defect within a domain only affects that domain's memory.

2) Leveraging different pre-existing detection mechanisms, such as stack canaries and domain violations.

We realize this idea by providing a C library, *Secure Domain Rewind and Discard (SDRaD)* for Linux on commodity 64-bit x86 processors with Protection Keys for Userspace (PKU). The library provides flexible APIs to support different compartmentalization schemes, such as protecting application integrity and confidentially. The developer can retrofit software applications with the secure rewinding mechanism by using them.

We evaluated our solution in three different use cases: Memcached, NGINX, and OpenSSL and concluded that it adds negligible overhead (2%–4%) in realistic multi-processing scenarios. Our solution can provide faster recovery time compared to container or process restart time. For example, in our Memcached setup with a 10GB database, a regular restart takes about 2 minutes, in-process rewinding takes only $3.5\mu s$. Also, our approach offers significant advantages with limiting the impact of malicious clients on other clients in a service-oriented application, without disrupting service.

One of the drawbacks of our solution is that retrofitting an application with SDRaD requires additional software development effort. Depending on the target application, this effort might be minimal. For example, we changed two source files in Memcached and added 484 new lines of wrapper code.

III. Friend or Foe? Foreign Functions in Rust

One of the shortcomings in Rust's foreign function interface is that it allows calling unsafe code inside Rust. In an earlier approach, unsafe code was invoked as a separate process with significant run-time overheads [9]. Also, several researchers propose heap isolation between the unsafe and safe blocks based on PKU by using compiler-based approaches [8], [10]. However, while both approaches can provide integrity in the memory safe area, they do not address the availability of software applications.

In ongoing work, we propose a solution, *SDRaD for Foreign Function Interfaces (SDRaD-FFI)*, to improve the availability and memory-safety guarantees of software written in Rust without excluding the use of legacy code. This is possible by protecting the integrity of the safe memory area from the unsafe area and allowing the safe area to perform alternate actions in case of memory violation. This requires lightweight in-process isolation between all safe and unsafe memory areas.

We will provide a Rust crate as a realization of our approach by leveraging our SDRaD C library. Moreover, we aim to provide easy-to-use annotations by leveraging Rust's macro expansion to hide SDRaD calls, argument and return value handling, and alternate actions in case of domain violations. SDRaD-FFI can support arbitrary argument passing between domains using different Rust serialization crates. We plan to evaluate different serialization crates and our solution in real-world use cases.

IV. Sustainability Evaluation

Providing well-defined components and isolation is important for sustaining security [15]. For example, isolation can limit memory corruption within a component. However, some approaches for compartmentalizing applications can be less environmentally sustainable than others. For example, conventional process isolation has high context-switching costs that increase resource utilization. Hardware-assisted in-process isolation, such as Memory Protection Keys (MPK) [13], [14] and CHERI [17], are potential solutions to provide lightweight isolation. It should be noted that CHERI requires specialized hardware. Implementing these solutions may require refactoring applications, which comes with software development costs. Another important issue is that while compartmentalizing applications into distinct domains is important for protecting application integrity and confidentially, providing availability is also important for environmental sustainability. Our solution is based on hardware-assisted in-process isolation by leveraging MPK; it is lightweight and incurs minimal runtime overhead. SDRaD APIs allow the developer to compartmentalize applications into distinct domains to provide confidentially and integrity. Also, the rewinding mechanism is important for application availability. While implementing SDRaD requires software development effort, we aim to reduce the effort for SDRaD-FFI with easy-to-use APIs.

Replication or diversification of software can decrease the likelihood of memory-related attacks and increase software longevity. This can result in over-provisioning hardware resources and is not environmentally friendly. Our solution supports fast recovery time without replication or diversification, and with only limited runtime overhead. In fact, SDRaD substantially reduces the time to recover from a fault. For example, in our Memcached setup, a regular restart takes about 2 minutes (which would violate 99.999% availability if there were three faults per year), while our in-process rewinding takes only $3.5\mu s$, allowing for more than $9 \cdot 10^7$ recoveries. Specifically in critical application scenarios, e.g., in telecommunications or smart grids, high levels of availability are normally achieved by means of redundancy, which our approach can alleviate. A thorough analysis of the potential impacts of our approach requires further life-cycle assessment approaches with a focus on environmental sustainability through energy efficiency [2], [7], but also economic and social dimensions [1], to be applied in a comprehensive case study from the above domains, which would also consider rebound effects [4]. Our results already show that in particular systems that involve software components such as Memcached, which need a lot of time to recover from failure, can benefit from SDRaD in terms of dependability and life-cycle sustainability.

V. Conclusion

We outline early research on applications and impact assessment of our *Secure Rewind and Discard of Isolated Domains* approach to improve software resilience and availability. We highlight ongoing work to improve on dependability properties of software written in memory-safe languages that rely on foreign/native library code. Importantly, our approach has the potential to also improve on aspects of environmental sustainability in critical software systems.

VI. Acknowledgments

This research is partially funded by the Research Fund KU Leuven, the Flemish Research Programme Cybersecurity, and the CyberExcellence programme of the Walloon Region, Belgium. This research has received funding under EU H2020 MSCA-ITN action 5GhOSTS, grant agreement no. 814035.

References

[1] Nelly Condori-Fernandez, Patricia Lago, Miguel R. Luaces, and Ángeles S. Places. An action research for improving the sustainability assessment framework instruments. *Sustainability*, 12(4):1682, 2020.

[2] Stefanos Georgiou, Stamatia Rizou, and Diomidis Spinellis. Software development lifecycle for energy efficiency: Techniques and tools. *ACM Comput. Surv.*, 52(4), aug 2019.

[3] Google Project Zero. 0day "in the wild" dataset, June 2022. Retrieved February 26, 2023 from https://web.archive.org/web/20230226000529/https://googleprojectzero.blogspot.com/p/0day.html.

[4] Cédric Gossart. Rebound effects and ICT: a review of the literature. *ICT innovations for sustainability*, pages 435–448, 2015.

[5] Daniel Gruss. Security: Can we afford to have it? can we afford not to have it? Talk at https://securityweek.at/2022/, September 2022.

[6] Merve Gülmez, Thomas Nyman, Christoph Baumann, and Jan Tobias Mühlberg. Rewind & Discard: Improving software resilience using isolated domains. To appear in DSN'23, 2023. A technical report is available at https://arxiv.org/pdf/2205.03205.pdf.

[7] Eva Kern, Lorenz M Hilty, Achim Guldner, Yuliyan V Maksimov, Andreas Filler, Jens Gröger, and Stefan Naumann. Sustainable software products—towards assessment criteria for resource and energy efficiency. *Future Generation Computer Systems*, 86:199–210, 2018.

[8] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. PKRU-Safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 132–148, New York, NY, USA, 2022.

[9] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *PLOS'17*, PLOS'17, page 51–57, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with XRust. In *ICSE '20*, ICSE '20, page 234–245, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[12] Jan Tobias Mühlberg. Sustaining security and safety in ICT: A quest for terminology, objectives, and limits. In *LIMITS'2022*, 2022.

[13] Soyeon Park, Sangho Lee, and Taesoo Kim. Memory protection keys: Facts, key extension perspectives, and discussions. *IEEE Security & Privacy*, pages 2–9, 2023.

[14] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ATC '19, page 241–254, Berkeley, CA, USA, 2019. USENIX Association.

[15] Andrew Paverd, Marcus Völp, Ferdinand Brasser, Matthias Schunter, Nadarajah Asokan, Ahmad-Reza Sadeghi, Paulo Esteves-Veríssimo, Andreas Steininger, and Thorsten Holz. Sustainable security & safety: Challenges and opportunities. In *4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[16] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 20–37, Washington, DC, USA, 2015. IEEE.