COMP 517: Endocert Midterm Report

Alexis Le Glaunec
Rice University

Senthil Rajasekaran
Rice University

Ziyang "Zion" Yang
Rice University

Abstract

The problem of proving real systems correct has begun to receive quite a bit of attention in the literature as formal methods tools have grown in complexity and power. These tools allow for verification of properties that is definitive as opposed to empirical testing-based methods which can only draw conclusions probalistically. New architectures, therefore, should be validated with these tools in order to increase trust and verify the principles of their designs. Recently, Im et. al. have introduced the endokernel architecture, which at a high level places a monitor inside each process in order to increase security. While the idea is promising, it is currently only validated through ad-hoc testing-based methods. This work aims to strengthen those claims of increased security by implementing a small-scale model that represents the key design principles of the endokernel architecture in Dafny and verifying security invariants on this model. By doing so, we are able to validate the design principles of the endokernel architecture independently from any specific implementation.

1 Introduction

Processes lie at the heart of the functionality of any operating system. Therefore, analyzing the failure tolerance / security of processes is fundamental to the proper functioning of modern computing systems. Specifically, it is important to isolate process from one another, as ideally the corruption of one process should not be affect other aspects of the computing system. Prior works have tried to improve the security of processes through software-based solutions in order to control the access of system resources at the user-level [20]. By controlling accesses at the user-space, the overhead of the system is significantly reduced as opposed to traditional kernel-based approaches [8, 12, 18]. Among these software-based approaches is the *endokernel* [9], which differs from previous work in that it fully virtualizes the system resources at the user-level, thereby providing a lightweight, non-bypassable solution to the problem of process isolation.

The endokernel is based on two fundamental ideas : (1) mapping the virtual machine to the traditional process level abstractions while (2) maintaining efficiency and extensibility to support customized protection abstractions. This allows the endokernel to prevent attacks to third-party software components that previous architectures struggled to address. As a result, the endokernel is more resilient to a wider array of attacks, such as the software supply-chain attack [2–4, 6, 7, 13, 16]. By providing non-bypassable memory isolation, the endokernel is able to defend against these kinds of attacks that have plagued previous architectures [19] by accessing unauthorized memory through file descriptors. If the endokernel is able to significantly improve the security of a system at the cost of a very modest overhead, then [] represents a very important idea in the evolution of operating system architectures.

As it currently stands, the endokernel has been verified through an ad-hoc, testing-based method. While this is certainly a robust method of testing systems, due to the complexity of this architecture testing and fuzzing based methods cannot reasonably cover all avenues of attack. Therefore, these methods fall short of a formal guarantee on the functionality of the endokernel. In order to provide such guarantees, formal method tools such as Coq [1], Isabelle [15], and Dafny [11] have been used to supply robust proofs of correctness for cutting-edge systems [10]. Through the use of these tools it is possible to prove security-based invariants in a rigorous way, increasing the trust and transparency of new architectures. The main goal of this paper is to apply formal method tools to validate the endokernel model by showing that the endorkernel architecture prevents one subprocess from interfering with another subprocess within the context of a larger process. This defines the threat model we are concerned with in this work, which is the scenario of a subprocess trying to access resources outside of its capabilities specified according to a security policy.

Of the tools discussed previously, we have chosen to use Dafny because it facilitates contract-based programming which is significantly easier than proof construction methods, such as Coq and Isabelle. Furthermore, the syntax of Dafny is closely related to C#, meaning that it shares more of a resemblance to the existing endokernel implementation written in C. By creating a small scale high level model (the strength of these tools limits the complexity of the models to be analyzed) of the endokernel architecture and proving security invariants in Dafny on this model, we are able to validate the endokernel model with respect to its claims of increased security. In order to perform our analysis, we assume the correctness of Dafny and that our implementation of the endokernel model is faithful to the design and intention of the implementation of the endokernel in [9]. The security of the endokernel will then be measured by incrementally adding properties on the system abstractions to the Dafny model and then verifying them. These properties will be scaled on quality by the severity of the threat they represent should they be violated.

2 Background

Our work is motivated by a long standing collaboration between the field of systems research and the field of formal verification aimed at proving critical systems correct []. In this section, we survey a few papers of particular note that we have found relevant to our methodology and design.

In 2021, Im et. al. [9] introduced the endokernel, which is the main object of our study. There are many approaches that have employed formal methods tools in order to verify the architecture of components of operating systems. We proceed with a brief survey of some relevant literature.

ExpressOS [14] formalizes a microkernel for mobile devices. In order to verify critical security invariants on implementations of these devices, `BOOGIE`-style tools (such as Dafny and VCC) are incorporated into the code. This method consists of adding code contracts to the code which are comprised of pre-conditions, post-conditions and assertions. Afterwards, an SMT-solver is used to prove that all contracts are respected. Ghost variables, i.e. variables with no semantics for the kernel, are added to facilitate the construction of the proofs. This approach is expected to take significantly less time than full completeness verification like [10] in `Isabelle`.

BesFS [17] aims to protect an application from tampering from a malicious OS. It consists of two parts: (1) a formal specification of the file-system interface that any OS should comply with and (2) a formally verified monitor of this OS file-system interface and encryption of application data using Intel SGX techniques. BesFS runs as a library linked to user-space applications. In the case of malicious OS behavior (defined as behavior that is deviant from policy specifications), BesFS is able to detect this deviation and abort the user application. Formalization is used as a means to (1) verify the specification of the OS API to ensure *safety* (i.e., capturing well-known attacks) and *completeness* (i.e, maintaining normal functionality) and (2) verify the implementation of BesFS.

Code-Level Model Checking in the Software Development Workflow [5] aims to verify the industrial library "aws-common". This is done by verifying the C code with CBMC, a bit precise bounded model checker for the C language. In CMBC specified pre-conditions and post-conditions of functions are verified through symbolic executions, allowing programmers to prove correctness properties about C code.

3 Methodology

The main contribution of this work is the creation of the high level model of the endokernel architecture in Dafny. We begin by providing an overview of the fundamental abstractions and functionalities of the model.

3.1 Overview of Modeling Principles

Intuitively, the endokernel architecture provides OS-style monitoring at the level of processes [9]. Our model is focused on the abstractions of the *endoprocess* and the *endokernel*. We believe that these two abstractions characterize the design and intent of the endokernel architecture, and they are defined as follows.

Definition 3.1 (Endoprocess)**.** *[9] The* endoprocess *is a protection domain nested within the address space of a process. Each endoprocess is associated with a segment of the process memory and a set of policy-defined capabilities and permissions.*

Definition 3.2 (Endokernel)**.** *[9] The* endokernel *is a monitor nested within the address space of a process, analogous to a kernel that monitors a process. The endokernel is then responsible for mediating access of system resources of the endoprocesses.*

In traditional OSes, a protection domain is defined by the pair $\langle process\_id, thread\_id \rangle$. The fundamental idea of the endokernel architecture is that a protection domain should be defined by the triple $\langle process\_id, thread\_id, endoprocess\_id \rangle$. This allows different instructions to be run at different permission and capability levels by executing them within different endoprocesses in accordance with the least-authority design principle. Each endoprocess then serves as a lightweight virtual machine in which instructions executed with similar permissions and capabilities can be run in isolation. The endokernel then serves as a monitor and mediator for the endoprocesses.

3.2 Model Implementation

The model we have implemented in Dafny is based on the UML shown in Figure 1, which demonstrates the basic abstractions needed to represent the endokernel architecture. The components of the UML were then implemented in Dafny, which can be found on GitHub. [1] In the Dafny model instructions, represented as strings, flow from a process to different endoprocesses (assigned by the endokernel) based on a user-specified policy of which permissions and capabilities the instructions should be run with. The endokernel creates endoprocesses and allocates these endoprocesses memory from the process memory space (which, at a higher level, is assigned by the kernel) as needed without letting endoprocesses interfere with each others' memory. Processes instructions are executed by threads that traverse many endoprocesses depending on which instruction the Program Counter (PC) is pointing to, as orchestrated by the endokernel. During traversal, threads update their Stack Pointer (SP) every time they switch between endoprocesses, which is denoted thread migration.

3.3 Instruction Flow

In this section, we discuss how instructions flow from processes to endoprocesses through endorkernels in our Dafny model. In order to achieve this, we demonstrate sections of the Dafny model code implementation and trace the behaviors of input instructions through the model.

We begin by exhibiting the implementations of the endokernel, endoprocess, and capability classes. These differ slightly from the UML diagram presented in Figure 1 due to minor implementation details.

The endokernel architecture is based on running different sets of instructions with different capabilities. It is therefore imperative to begin with the implementation of the capability class.

```
class Capability {
    var subspace: seq<int>
    var stack: seq<int>
    var files: seq<string>
    var entries: seq<string>
}
```

The capability class represents a set of permissions with which to run certain instructions. These permissions depend on the policy enforced by the endokernel, which is implemented in the following way.

```
class Endokernel {
    var capabilities: map<Capability, Endoprocess>
    var endoprocesses: map<int, Endoprocess>
    var instructionMap: map<string, Capability>
    var nextPid:int;
    var kernel:Kernel;
}
```

[1]https://github.com/alexis51151/comp517-project

The endokernel has an instructionMap which associates instructions from the instruction set, represented as strings that represent C code, to a capability. It is able to trap instructions and send them to the relevant endoprocess associated with the policy-assigned capability through the use of the Capabilities map. The instructions should then run in an endoprocess, which is implemented in the following way.

```
class Endoprocess {
    var id:int;
    var memorySpace:seq<int>
    var instructions: string
    var endokernel:Endokernel;
}
```

The relevant endoprocess then runs the instruction in its memorySpace. It can then communicate back to the endokernel in order to execute the instructions elsewhere.

The flow of the instruction `write(0, "toto", 4)` is shown in the following Figure 2. It begins at ① when the process issues a new write instruction to the file descriptor 0 for value *"toto"*. This process instruction is then trapped in step ② by the endokernel nested within the process. In step ③, the endokernel executes a look up in its *instructionMap* to look for the policy regarding the *write* instruction. Then, if there already exists an endoprocess with this capability, the instruction is forwarded to that endoprocess in step ④. Otherwise, a new endoprocess with the necessary capability is created to execute the instruction. The relevant endoprocess executes the instruction which is then trapped back in ⑤ by the endoprocess to ensure that the endoprocess is not violating its capabilities. Finally, if the endoprocess has the proper permissions, the syscall is forwarded to the kernel in ⑥. Running this is the implemented model produces the following trace, demonstrating the desired flow of instructions in the model implementation.

3.4 Verification

As mentioned previously, the model will be tested for various security properties that both represent significant security threats and were of interest to the authors of the endokernel architecture paper [9]. A preliminary listing of these properties follows.

Property 3.1 (Endoprocess Isolation)**.** *At a high level, endprocesses should not be allowed to interfere with each other. Each endoprocess should be able to have access to its own resources according to its assigned capabilities, and should not be able to interfere with the capabilities and/or resources of another endoprocess.*

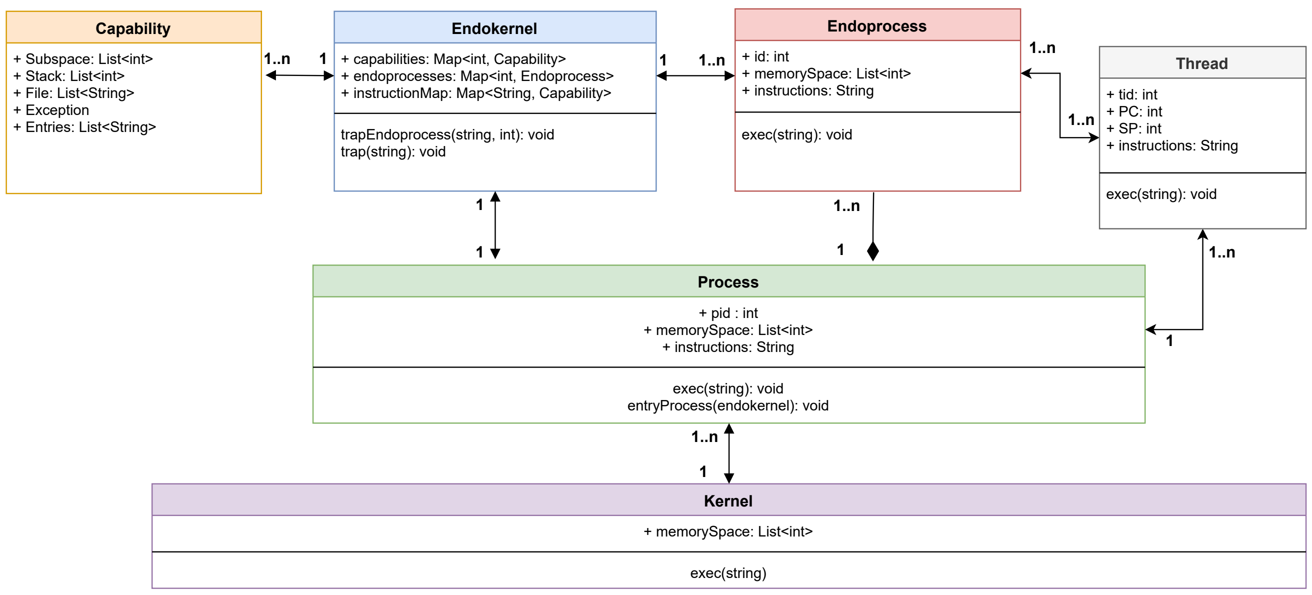

Figure 1: UML Diagram of the Endokernel Architecture representing well-known OS abstractions: Kernel, Process, Thread, Capability and two new abstractions: the Endokernel, a monitor nested within a process, and the Endprocess, a protection domain nested within its process adress space.

Should this property be violated, the endokernel architecture is broken at a fundamental level. The design of the endokernel architecture is predicated on the fact that the endokernel is able to run different instructions with different capabilities, and so a violation of this property is a violation of the most basic design principle of the architecture.

Property 3.2 (Complete Mediation)**.** *All instructions should go through the endokernel for monitoring.*

If there are some instructions that are not monitored, this represents a significant security threat for the process and architecture as a whole. For example, an instruction in one endoprocess could accidentally crash the whole program by throwing a divide-by-zero error. The endokernel architecture provides added security by monitoring instructions, and so if there are instructions that are unmonitored the system is vulnerable.

Property 3.3 (Nexpoline)**.** *The endokernel traps and executes all system calls on behalf of the kernel.*

For system calls, which constitute a subset of all possible instructions, the endokernel architecture requires that they be made through the endokernel itself. This allows the endokernel to trap and execute instructions that interact with the kernel. Unauthorized accesses to the kernel could have severe consequences for the function of the computing system as a whole. For example, a malicious third-party library could power the whole machine off with a malicious system call.

These properties will be formulated as preconditions and postconditions in Dafny which will be then be verified through Dafny's contract verification tools.

4 Evaluation and Results

As of this midterm report, the model has been implemented but not verified with respect to the outlined properties. Verification with Dafny is a challenging and time consuming process as we need to explicitly declare what each method modifies in a consistent way and then prove invariants on the methods. We expect this family of tasks to provide a good amount of work for the rest of our project. We plan to focus on proving the outlined properties (and potentially more) for different system calls like `write` or `read`. Therefore, there are no evaluations or results to present at this time.

As a further note, the evaluation may also be extended to include the time it takes for different properties to be proven, as lower runtimes correlate to higher quality proofs.

5 Discussion

This work is part of a long standing collaboration between the fields of system design and formal methods based around proving real systems correct. Our contributions will be twofold:

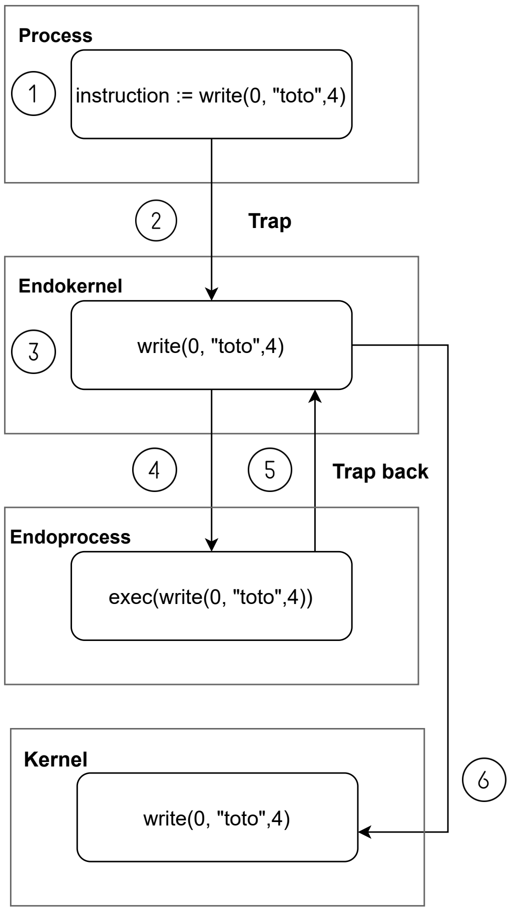

Figure 2: Instructions flow from the process, where they are initialized to the endokernel, where they are trapped and sent to the relevant endoprocess. The instructions are then trapped back by the endokernel from the endoprocess and sent to the kernel for execution.

Implementation of a High Level Dafny Model We have implemented the fundamental design principles behind the endokernel architecture in Dafny. This small scale model was created from scratch in order to reason about the high level security invariants that must hold for the endokernel architecture to function properly. By using a tool like Dafny, we are able to prove these invariants in a completely robust way, unlike ad-hoc, empirical, testing-based methods. While testing-based methods are able to draw conclusions about the system with high probability as more and more testing is done, formal methods tools are able to prove properties definitively in one shot. Furthermore, the creation of this model allows us to reason about the fundamental principles of the endokernel architecture independently from any specific implementation.

Verification of Security Principles on the Model The future work of this project involves utilizing the Dafny model in order to explicitly verify security-based properties of system calls in the endokernel architecture. As the semester progresses further, we plan on specifying these security principles through pre and post conditions and then verify them using Dafny. This is made possible by the fact that we have created our model from the ground up in Dafny.

```
1: Executing instruction write in Process
----------------------------------------------
2: Trapping instruction write in Endokernel
----------------------------------------------
4: Executing instruction write in Endoprocess
----------------------------------------------
5: Trapping back instruction write in Endokernel
----------------------------------------------
6: Executing instruction write in Kernel
```

Figure 3: Instruction flow trace for instruction `write(0, "toto", 4)`.

Once we have verified these security principles, it will be possible to evaluate the design of the endokernel itself with respect to its claims of added security. Should these results be robust enough, we will be able to increase trust in the principles of the endokernel architecture and hopefully encourage its commercial use.

Extensions of our model, such as ones based on lower level abstractions will better approximate the endokernel architecture, making this a natural avenue for future work. As the model is expanded and made more robust, it becomes possible to both express new security invariants with respect to the model and verify them. Since the endokernel itself consists of relatively few lines of code, it is not unreasonable to imagine an entire endokernel implementation in Dafny created with the purpose of verification.

Furthermore, we note that the approach taken in this work is relatively novel. As shown in the background, a more common approach towards the problem of verifying security invariants in real systems includes focusing on a smaller part of the overall code and embedding a verified version of this into a larger system. Therefore, our method could be thought of being “top-down", in which a representation of the entire system is considered, as opposed to “bottom-up", in which individual pieces of the system are considered separately and them unified. While the top-down approach is able to approach the problem of proving security invariants correct from the perspective of the invariants themselves, the bottom-up approach takes the perspective of the implementation itself by considering its components. A promising avenue of future work would then be the application of such bottom up techniques to prove critical components of the endokernel correct, and it seems this would be a more practical but less theoretical approach towards evaluating the endokernel architecture.

Overall, we believe in the endokernel architecture and believe that it will hold up to scrutiny when tested. Therefore, we are enthusiastic with regards to the prospect of work designed to evaluate this architecture.

6 Conclusion

Our key contribution is the implementation of the Dafny model. As the semester progresses, we plan on verifying properties on this model using Dafny's contract verification tools. This will allow us to validate the principles behind the endokernel architecture.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[2] Jon Boyens, Celia Paulsen, Rama Moorthy, Nadya Bartol, and Stephanie A Shankles. Supply chain risk management practices for federal information systems and organizations. *NIST Special Publication*, 800(161):32, 2015.

[3] Hugh Boyes. Cybersecurity and cyber-resilient supply chains. *Technology Innovation Management Review*, 5(4):28, 2015.

[4] Sandor Boyson. Cyber supply chain risk management: Revolutionizing the strategic control of critical it systems. *Technovation*, 34(7):342–353, 2014.

[5] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R Tuttle. Code-level model checking in the software development workflow. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 11–20. IEEE, 2020.

[6] Jason K Deane, Cliff T Ragsdale, Terry R Rakes, and Loren Paul Rees. Managing supply chain risk and disruption from it security incidents. *Operations Management Research*, 2(1):4–12, 2009.

[7] Abhijeet Ghadge, Maximilian Weiß, Nigel D Caldwell, and Richard Wilding. Managing cyber risk in supply chains: A review and research agenda. *Supply Chain Management: An International Journal*, 2019.

[8] Terry Ching-Hsiang Hsu, Kevin J. Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 393–405. ACM, 2016.

[9] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. The endokernel: Fast, secure, and programmable subprocess virtualization. *CoRR*, abs/2108.03705, 2021.

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[11] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[12] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An OS abstraction for safety and performance. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 49–64. USENIX Association, 2016.

[13] Tianbo Lu, Xiaobo Guo, Bing Xu, Lingling Zhao, Yong Peng, and Hongyu Yang. Next big thing in big data: the security of the ict supply chain. In *2013 International Conference on Social Computing*, pages 1066–1073. IEEE, 2013.

[14] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. *SIGPLAN Not.*, 48(4):293–304, March 2013.

[15] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[16] Melinda Reed, John F Miller, and Paul Popick. Supply chain attack patterns: Framework and catalog. *Office of the Deputy Assistant Secretary of Defense for Systems Engineering*, 2014.

[17] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. Besfs: A POSIX filesystem for enclaves with a mechanized safety proof. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 523–540. USENIX Association, August 2020.

[18] Zahra Tarkhani and Anil Madhavapeddy. Enclave-aware compartmentalization and secure sharing with sirius. *arXiv preprint arXiv:2009.01869*, 2020.

[19] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[20] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.