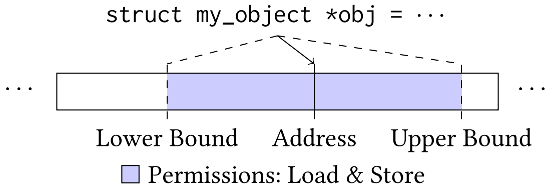
Library-based Compartmentalisation on CHERI

Dapeng Gao
University of Cambridge
United Kingdom
dapeng.gao@cl.cam.ac.uk

Robert N. M. Watson
University of Cambridge
United Kingdom
robert.watson@cl.cam.ac.uk

1 Introduction

Existing software compartmentalisation solutions tend to incur considerable performance or refactoring costs. And it seems difficult, at least on conventional hardware, to reach the sweet spot of simultaneously having *low overhead*, *wide applicability*, and *easy migration*. This article presents early results indicating that such a sweet spot can be reached on a CHERI-enabled architecture. Our key contributions are a) a *compartmentalisation model* based on dynamic linkage and b) a prototype implementation of the model on Morello, a high-performance AArch64 CPU extended with CHERI [7].



CHERI extends existing instruction set architectures into *capability systems* where pointers contain bounds and permissions in addition to an integer memory address [8]. The above diagram illustrates how a CHERI architecture represents a C pointer, which can now only be used to access a limited range of memory with restricted privileges.

We summarised the following design principles which proved useful in guiding our research.

1. Prioritise compatibility—programs should 'just work'.
2. Allow for incremental adoption of security policies.
3. Compartmentalisation should be nearly transparent to the programmer during development and debugging.
4. The compartmentalisation mechanisms should build upon concepts familiar to the programmer.

2 Programming model

We understand a 'compartment' to be a collection of code within a program that share a set of privileges with regard to how they can interact with the operating system and other compartments of the program. Importantly, a compartment must not be construed as a 'thread', 'task', or 'process' that represents some *execution* of its code with its own set of privileges. Rather, *all* executions of the code within a compartment share the same privileges.

Library-based software compartmentalisation is, simply put, a programming model that treats each dynamic library used by a program as a separate compartment. It is largely incremental upon the C/C++ programming model, which is a deliberate design choice that prioritises compatibility with existing code to make migration efforts easier. More concretely, programmers can expect most existing C/C++ software to continue to function when library-based software compartmentalisation is enabled with an 'all-permissive' security policy. This security policy can then be tightened to restrict what each library is permitted to do, such as:

1. What functions it is allowed to call,
2. What global variables it is allowed to access,
3. What system calls it is allowed to make,
4. What signals it is allowed to handle, and
5. What non-standard control flow (e.g., `setjmp`/`longjmp` and C++ exceptions) it is allowed to carry out.

Why is the dynamic library chosen as the unit of compartmentalisation? We hypothesise that compartmentalising at library boundaries sufficiently minimises the privileges granted to each piece of untrusted code to contain the damage of potential security vulnerabilities. This is because most modern software comprise a main executable linked against several dynamic libraries, each of which typically serving a narrow set of purposes such as image decoding, language interpreter, and cryptography. Moreover, libraries are also a *convenient* target to compartmentalise. Finer-grained compartmentalisation, where compartment boundaries can be drawn *within* libraries, is not supported due to the high migration costs it would entail—non-trivial modifications to the source code would likely be needed to demarcate compartment boundaries [1, 2, 4, 6]. However, such modifications can potentially be automated in the future to make more granular compartmentalisation feasible at scale.

3 Threat model

The discussion up to this point has only referred to 'untrusted code' informally. While it would certainly be ideal to be able to claim that a model of compartmentalisation can achieve the desired security guarantees for arbitrary binaries, a trade-off between feasibility, performance, and security likely exists in practice. This trade-off induces a wide range of *threat models*: this section explains the one assumed by the present work.

The key assumption of our threat model is that the ELF files containing the untrusted dynamic libraries are well-formed. This means that the file not only conforms to the ELF format, but the metadata within also follow appropriate security standards. For example, the virtual memory mappings specified in the segment header should have correct permissions, and the dynamic symbol table should not export function or data symbols that are intended to be private.

In practice, executables and libraries that are built from source by a trusted toolchain will satisfy the above requirements. Hence, the toolchain used for creating these libraries is considered part of the trusted computing base (TCB).

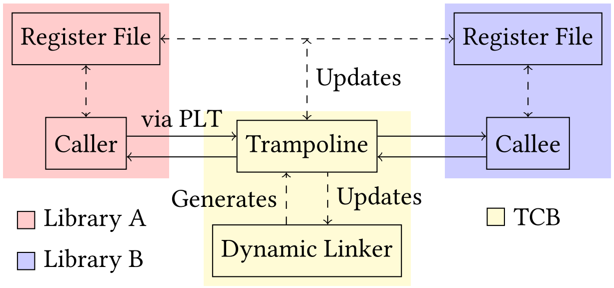
By adopting this threat model, we intend to mitigate vulnerabilities in the source code of untrusted dynamic libraries but do not aim to tackle threats that use maliciously-crafted ELF files as an attack vector. However, it is conceivable to design a program that can statically check untrusted ELF files against a pre-specified list of properties to gain more assurance about their security.

4 Implementation overview

Because libraries can only gain access to external resources through function or data symbols resolved by the dynamic linker, we identify dynamic linkage as the ideal place to incorporate compartmentalisation into the existing C/C++ programming model. Moreover, since the dynamic linker is the first piece of user-space code to execute when a dynamically-linked program is launched, it is naturally in a position of trust and hence chosen to be part of the TCB.

More concretely, when resolving a function symbol, instead of directly writing the target function's address into the requesting library's GOT, the dynamic linker generates a *trampoline* and inserts the trampoline's address into the GOT entry. Consequently, when that function is called from an external library, the PLT transfers control to the trampoline, which performs a *domain transition* before redirecting control to the actual function. And when the function returns, it returns back to the trampoline, which reverses the domain transition before actually returning control to the caller.

The domain transition primarily enforces two properties: *well-bracketed control flow* and *stack safety*, of which Georges et al. [3] provided a detailed formalisation. The former is enforced by keeping an authoritative chain of return addresses in a *trusted stack* accessible only to the TCB, the latter by assigning a distinct stack to each compartment which is switched by the trampoline during domain transition.



The diagram above illustrates the interaction between various elements during a inter-library function call when compartmentalisation is enabled. The trampoline intercepts the control flow both during call and return, but is intended to be completely transparent to the software. It pushes the return address to the trusted stack and updates the content of the register file, including switching the stack pointer.

Several complications arise from this design. Non-standard control flow such as `setjmp`/`longjmp` and C++ exceptions need special handling to function correctly. The calling convention needs to be modified so that no function arguments are passed implicitly via the stack. The threading library needs to be aware of multiple stacks per thread. And the special GOT entries belonging each library that point to the dynamic linker's internal data structures need to be secured against corruption from untrusted code.

Initial performance benchmarks on an earlier version of the prototype see a low single-digit percentage of overhead.

5 Next steps

The current prototype provides no security guarantees and is intended to serve as a foundation for researching and implementing various policies. The following next steps have been identified as essential for a mature solution.

Compartment-aware debuggers In particular, it should be able to unwind call frames across multiple stacks.

Security policy description language Security policies should be expressible in a concise language, which may then be embedded at compile-time into the main executable to be used by the linker.

Fine-grained system call restriction It is often only useful when system calls can be restricted depending the arguments passed to them. In these cases, a mechanism for intercepting system calls and filtering them based on the arguments passed needs to be developed.

Register clearing Unused caller-saved registers can contain sensitive data that may be unintentionally leaked to the callee. They should thus be cleared before transitioning to a new domain. However, information about which registers need to be cleared can only be known at compile-time. Therefore, the compiler should generate code to clear such registers for potentially domain-crossing calls.

Function pointers Currently, when a pointer to a static function is passed to another compartment and called there, a domain transition back to the original compartment is not triggered, resulting in the function executing with incorrect privileges. The compiler should detect this type of usage and force such function symbols to be dynamically relocated and hence interposed by a trampoline.

Compartment interface vulnerabilities The compartmentalisation model does not mitigate vulnerabilities caused by mis-designed APIs (e.g., failure to validate input data), as identified by Lefeuvre et al. [5]. In addition, it is unclear how memory buffers shared between compartments can be secured against corruption without resorting to relatively costly message-passing or MMU-based solutions.

Acknowledgments

Approved for public release; distribution is unlimited. This work is sponsored by Innovate UK project 105694, 'Digital Security by Design (DSbD) Technology Platform Prototype', Office of Naval Research (ONR) Contract No. N00014-22-1-2463 ('SoftWare Integrated with Secure Hardware (SWISH)'), and the Croucher Foundation. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

References

[1] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)* (San Francisco, CA, Apr. 2008), USENIX Association.

[2] Brumley, D., and Song, D. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th USENIX Security Symposium (USENIX Security 04)* (San Diego, CA, Aug. 2004), USENIX Association.

[3] Georges, A. L., Trieu, A., and Birkedal, L. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *Proceedings of the ACM on Programming Languages 6*, OOPSLA1 (Apr. 2022), 1–30.

[4] Kilpatrick, D. Privman: A Library for Partitioning Applications. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)* (San Antonio, TX, June 2003), USENIX Association.

[5] Lefeuvre, H., Bădoiu, V.-A., Chen, Y., Huici, F., Dautenhahn, N., and Olivier, P. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *Proceedings 2023 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2023), Internet Society.

[6] Provos, N., Friedl, M., and Honeyman, P. Preventing Privilege Escalation. In *12th USENIX Security Symposium (USENIX Security 03)* (Washington, D.C., Aug. 2003), USENIX Association.

[7] Watson, R. N. M., Moore, S. W., Sewell, P., and Neumann, P. G. An Introduction to CHERI. Tech. rep., Computer Laboratory, University of Cambridge.

[8] Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., Davis, B., Eisen, L., Filardo, N. W., Grisenthwaite, R., Joannou, A., Laurie, B., Markettos, A. T., Moore, S. W., Murdoch, S. J., Nienhuis, K., Norton, R., Richardson, A., Rugg, P., Sewell, P., Son, S., and Xia, H. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Tech. rep., Computer Laboratory, University of Cambridge.