FIDES: Hardware-assisted Compartments for Securing Functional Programs

ANONYMOUS AUTHOR(S)

Two major causes for the rapidly increasing threat of cyber-attacks are unsafe languages like C and C++ and a monolithic software architecture that combines code with varied security expectations without isolation. To counter these critical issues, developers are migrating to memory-safe languages like OCaml and Rust and employing software compartmentalization techniques that reduce the attack surface. Current compartment schemes are designed specifically for C and C++, and do not accommodate language features found in high-level programming languages, such as exceptions, first-class functions, tail-call optimisation, and garbage collection. That said, even with the advent of memory-safe languages, developers must still rely on legacy third-party libraries written in C and C++. This requires re-imagining the compartment schemes to work seamlessly in the presence of both unsafe and safe language codes.

This paper proposes Fides, a novel hardware-enabled compartment scheme designed for high-level, memory-safe, functional programming languages targeting resource-constrained embedded systems. Fides creates code compartments with custom compiler and hardware extensions. It leverages the language-level safety guarantees of a memory-safe language to enforce fine-grained data sharing across compartments. Fides' compartment scheme supports essential functional programming features like tail-call optimisation and higher-order functions. To permit mixed-language applications, Fides extends C with hardware-assisted fat pointers to preserve the guarantees of the compartment scheme.

Fides is realized by extending a RISC-V processor to support compartments in OCaml. We illustrate our technique by implementing Fides to secure mixed-language MirageOS unikernel applications and demonstrate a prototype on FPGA. Our results show that Fides executes OCaml code with no additional performance penalty, while the C code is secure but pays a small penalty to preserve the guarantees of the compartment scheme.

CCS Concepts: • **Security and privacy → Embedded systems security**; **Software security engineering**.

Additional Key Words and Phrases: Memory Isolation, Privilege Separation, LLVM, Compartmentalization, OCaml, Unikernels, Fat pointers

ACM Reference Format:
Anonymous Author(s). 2024. FIDES: Hardware-assisted Compartments for Securing Functional Programs. 1, 1 (July 2024), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Cyber attacks have been rapidly increasing in recent years. While several factors contribute to this growth, two prominent causes are the widespread use of C and C++ and the monolithic design of applications. The lack of built-in memory safety in C and C++ can lead to critical security vulnerabilities, making these programming languages one of the most insecure [20, 51]. Monolithic software design that mixes code with different security guarantees in the same address space makes every vulnerability dangerous, potentially compromising the entire system. For example, a vulnerability in an image processing library like `libpng` (CVE-2020-35511 [40]) used by a banking

ACM XXXX-XXXX/2024/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

application might be exploited by an attacker to compromise the entire application, leading to the loss of critical financial data. Even in a safe language, malicious third-party libraries from popular package ecosystems may steal sensitive data such as passwords and private keys [7, 8].

Two approaches often adopted to address these security problems are memory-safe programming languages and software compartments. Safe[1] languages, such as OCaml, Rust, Java, JavaScript and Python, rely on the compiler to guarantee the absence of memory safety vulnerabilities. Major open source projects, such as Linux [48] and Mozilla Firefox [37], are slowly transitioning towards safe programming languages. The other approach is software compartments [9, 15, 16, 25, 30, 31, 46, 56], which provide *intra-process* vulnerability isolation by partitioning software into isolated components where each component has just the required privileges to execute [45]. Compartments permit sensitive code to be isolated from the rest of the application, limiting the impact of a potential vulnerability in an unsafe language or a malicious third-party library [7, 8]. In a typical compartment scheme, each isolated component is defined by a code region called *code compartment* and a data region called *data compartment*. Access policies limit the control flow between functions to only those whitelisted and ensure that every memory access is within a predefined data compartment.

In practice, both methods encounter difficulties. The main obstacle in moving to a safer programming language is that large amounts of C/C++ code have been in use and working well for many years. An overnight transition to a safe language is, therefore, not practical. Organizations approach the shift gradually, progressively integrating a safer language into the application's existing codebase. This leads to applications that have safe languages mixed with unsafe C/C++ code in the same codebase. For example, the Firefox browser has around 40% C/Assembly/C++ code, 11% Rust code, and the remaining primarily consists of HTML and Javascript [36]. These mixed language codebases undermine the benefits of the safe language because of the memory vulnerabilities still present in the unsafe code [34].

As for software compartments, most compartment schemes are designed specifically for C and C++ software. A key challenge is that the absence of memory safety in C/C++ makes the design of data compartments challenging. For example, sharing a data structure across two compartments can be done either by (a) relaxing the compartment policy so that both compartments can access the shared data or (b) copying the data from one compartment to the other. While the former results in larger attack surfaces, the latter increases overheads and modifies program semantics (sharing references versus copies). Furthermore, to share a data structure across two compartments and get the correct semantics, we would require the programmers to have knowledge of the compartment scheme. Hence, it is often impractical to modify legacy code to suit a bespoke compartmentalization scheme.

So far, the compartmentalization and the transition to safe programming languages have been viewed in isolation. We observe that we can merge both approaches, complementing each other to arrive at a more robust security model. We leverage the memory safety guarantees of safe language to simplify the compartment design. Intra-process compartments hold the promise of (a) isolating unsafe C/C++ code from safe code and (b) isolating untrusted third-party libraries from sensitive parts of the program. Designing such a pragmatic compartment scheme for safe languages, however, remains challenging due to the following reasons:

- **C1** Applications developed in safe languages often link against legacy C and C++ libraries. Vulnerabilities in such libraries compromise the safety of the entire application. Therefore, any compartment scheme designed for safe languages should also accommodate the challenges of linking against C and C++.

[1] We write "safe" and "unsafe" languages to mean "memory-safe" and "memory-unsafe" languages, respectively.

- **C2** Higher-order functions (HoFs) and function closures are widely used in functional programming languages like OCaml and are becoming mainstream in languages like Python and Java. Current compartment schemes are too rigid and cannot handle them efficiently. For example, given that function closures may be allocated in one compartment and executed in another, care needs to be taken to accommodate both code and data compartmentalization of closures.
- **C3** High-level languages include language features (such as exceptions) and compilation techniques (such as tail call optimisation) that have complex control-flow characteristics. Extant compartmentalization techniques designed for C do not accommodate these features, making them unsuitable for high-level languages.

We propose FIDES, a fine-grained compartment scheme that provides isolation at function-level granularity, designed for applications that use untrusted modules and mix safe and unsafe languages within the same program. In this work, we use OCaml as the safe language and C as the unsafe language in our implementation of FIDES. In addition to the source code, FIDES requires a *policy file* that assigns functions to compartments, defines inter-compartment access policies, and specifies distinguished functions as valid entry points into compartments. All control flow within a compartment is unrestricted. However, control can enter a target compartment from another compartment only if the access policy permits it and only through valid entry points. FIDES uses hardware assistance to mediate control flow between the functions belonging to different compartments.

For data compartments, unlike prior work [46, 54] that utilise OS or MMU support for coarse-grained compartmentalization (often at page-level granularity), FIDES offers fine-grained data compartments at byte-level granularity. FIDES achieves this by building on top of memory safety guarantees of the safe language. A type-safe OCaml program is memory-safe. Only data that is accessible to an OCaml function is what is transitively reachable from its locals, function arguments and globals. To ensure that this property is preserved for an application that mixes OCaml and C and to bring the same guarantees to C (challenge **C1**), FIDES uses spatial and temporal memory safe C with hardware-assisted fat pointers [11, 13, 38, 39, 55, 60]. Fat pointers are known to have appreciable performance costs in practice. Luckily, we can avoid fat pointers for OCaml since the language ensures memory safety while paying attention to the foreign function interface (FFI) boundary between OCaml and C.

To address **C2**, FIDES introduces the notion of *fluid compartments* that facilitates flexible compartment strategies to securely share closures between compartments without compromising security guarantees. We design our compartment scheme to preserve tail-call optimisation and support exceptions, thus addressing **C3**.

Our approach is developer-friendly. Apart from a few unsupported language features in the unsafe language (§2.1), FIDES neither requires code changes nor changes the semantics of parameter passing for inter-compartment function calls. This allows us to readily use the large ecosystem of available libraries, including the MirageOS [33] library operating system. FIDES allows compartment access policies to be defined separately from the source code. This permits an expert security engineer to compartmentalize an application which may include untrusted third-party code. Since our compartmentalization does not rely on OS or MMU, it is suitable for constrained embedded systems that support neither. FIDES compiles OCaml- and C-based applications to run *baremetal* on a modified RISC-V [23] processor with two new instructions – (1) checkcap that enforces code compartments, and (2) val that enforces data compartments in C.

Our contributions are as follows:

- We present Fides, a fine-grained compartment scheme designed for applications that mix safe and unsafe language. Fides supports high-level language features such as higher-order functions, tail calls and exceptions.
- We present a formal operational model of Fides, inspired by RISC-V, to prove that Fides preserves the security guarantees provided by compartmentalization (§4).
- We present an implementation of Fides on a modified Shakti RISC-V processor [14] that executes baremetal MirageOS [33] unikernels (§5). Fides builds upon Shakti-MS [11], which provides spatial and temporal memory safety to C.
- We demonstrate the effectiveness of Fides with a security-critical electronic voting machine (EVM) application (§3). Our evaluation of Fides on the Xilinx Artix-7 AC701 FPGA [57] shows that Fides offers an attractive security-performance tradeoff (§6).

2 Threat model

In this section, we list the assumptions and limitations of Fides and describe the attack model.

2.1 Assumptions and limitations

Fides permits applications to be built with a mix of safe (OCaml) and unsafe (C) code. The application may contain untrusted third-party libraries, which may be malicious. The attacker has access to the entire source code of the application and has full knowledge of the internals of Fides and the compartment access policy. Hence, the attacker may exploit programmer-introduced bugs in the source code. To support fat pointers, on the C side, Fides does not support casting integers to pointers, unions with pointer fields and variadic arguments. We assume that the application is compiled using the Fides OCaml and C compilers, which are assumed to be correct. Fides supports hand-written assembly, but we trust this code to be correct. We also assume correct any use of the `Obj` module in OCaml that permits unsafe access to the OCaml heap. We assume that the Fides executable is a statically-linked binary that cannot be tampered with. Hardware attacks like Rowhammer [26], fault attacks [4], and side-channel attacks [32] are beyond the scope.

2.2 Attacks

Despite our assumptions and limitations, an application that mixes OCaml and C leaves open many attack vectors. Table 1 lists the major vulnerability classes present in a C + OCaml codebase.

2.2.1 Memory vulnerability. Since C does not offer memory safety, the C code can read and write to arbitrary parts of the OCaml heap and stack. Given that the attacker has full knowledge of the application's source code, they may craft an attack by writing to security-critical data in memory, leading to leaking information [49].

```
let admin_flag = ref false
...
if (admin_flag) (* do privileged operation *)
```

In the code above, the attacker may use a memory error-based CWE (Table 1), to update `admin_flag` to `true` to perform privileged operations. Appendix A in the supplementary material presents the source code for an attack that uses an out-of-bounds write to update the `admin_flag`. Making C memory safe helps thwart memory error-based CWEs. Fides thwarts this attack with the help of hardware-assisted fat pointers for C. Our implementation of Fides builds upon Shakti-MS [11], which provides spatial and temporal memory safety for C.

Table 1. Common Weakness Enumeration(CWEs) [10] that FIDES fully address or significantly weaken the damage.

CWE	C+OCaml	Memory-safe C + OCaml	FIDES
Memory error based CWEs			
CWE-415: Double Free	◐	🟢	🟢
CWE-416: Use after free	◐	🟢	🟢
CWE-125: Out-of-bounds read	◐	🟢	🟢
CWE-787: Out-of-bounds write	◐	🟢	🟢
CWE-121: Stack-based buffer overflow	◐	🟢	🟢
CWE-124: Buffer underwrite (buffer underflow)	◐	🟢	🟢
CWE-123: Write-what-where condition	◐	🟢	🟢
CWE-122: Heap-based buffer overflow	◐	🟢	🟢
CWE-562: Return of stack variable address	◐	🟢	🟢
FFI interactions	🔴	🟢	🟢
Privilige isolation based CWEs			
CWE-653:Improper isolation or compartmentalization	🔴	🔴	🟢
CWE-250:Execution with unnecessary privileges	🔴	🔴	🟢
CWE-441:Unintended proxy or intermediary (confused deputy)	🔴	🔴	🟢
CWE-1125: Excessive attack surface	🔴	🔴	🟢
CWE-767: Access to critical private variable via public method	🔴	🔴	🟢
CWE-691: Insufficient control flow management	🔴	🔴	🟢

🔴 : not mitigated | ◐ : partially mitigated only in OCaml codebase | 🟢 : Mitigated

2.2.2 Isolation. Our aim is to build secure MirageOS unikernels [33] for embedded systems. Unikernels combine application and OS code into a single-address-space executable. In particular, MirageOS unikernels offer no privilege separation mechanisms such as user and kernel modes, process abstractions, etc. While OCaml provides strong abstraction boundaries through modules and signatures, these abstraction boundaries may be defeated by C code, even with memory safety and the assumption that pointers cannot be forged. One attack vector is function closures, which are represented as objects on the heap that contain the code pointer and the environment. For example, consider the snippet below.

```
value *callback_sum = caml_named_value("sum");
value *callback_leak = caml_named_value("leak");
Store_field (*callback_sum, 0, Field(*callback_leak, 0));
```

Here, the C code accesses OCaml callbacks named `sum` and `leak` and overwrites the code pointer in the `sum` closure with that of the `leak` function. Importantly, the write is within the bounds of the `sum` closure, and hence, spatial memory safety is not enough to prevent this attack. Any subsequent calls to `sum`, including on the OCaml side, will now be subverted to the `leak` function. Appendix B in the supplementary material shows the entire working example. FIDES provides the compartment mechanism for specifying and restricting such unintended control flow in the program, helping prevent control flow subversion. We shall discuss the details of the mechanisms in the next section.

3 Case study: An EVM with FIDES

In this section, we illustrate the power of FIDES by implementing an electronic voting machine (EVM) application as a MirageOS unikernel. We also show how FIDES addresses the challenges with supporting compartments in expressive high-level languages.

Table 2. Compartments in EVM

ID	Name	Description
CP1	main-menu	Handles main menu and drives the EVM application.
CP2	admin	All the code which requires election official privilege.
CP3	crypto	C implementation of cryptographic libraries with OCaml wrappers.
CP4	votes-handler	Performs voter validation. Reads the plaintext vote from the user, encrypts using the crypto library, and stores it in the memory-mapped `votes table`. Contains all code handling unencrypted votes. **Sensitive.**
CP5	helper	All code that neither deals with votes in plaintext nor requires election official privilege, such as the code for the helper menu, time and display modules. Contains third-party libraries including C code. **Untrusted.**
CP6	ocaml-stdlib	OCaml standard library and the OCaml runtime.

3.1 Securing the EVM with Fides

Our EVM is an offline, embedded device that only runs the EVM application. The machine has an electronic display that lists the candidates and uses physical buttons to accept inputs. For each voter, the software validates the voter ID against a stored list of IDs and verifies that a vote has not already been cast for that ID. The vote is then read from the user, encrypted, and stored in the device. After all the votes are cast, the election official locks the EVM application from accepting further votes. On the counting day, the election official inputs their authorization key to decrypt and count the votes. The encryption is performed using an AES implementation in C.

Our goal is to prevent invalid votes, double voting, and leaking votes before the counting day. We use Fides to secure the EVM application and achieve its security goals. The high-level design of the EVM application is shown in Figure 1. We compartmentalize the application based on whether it requires an election official's credential to access stored votes. The six compartments, labelled **CP1** to **CP6**, are described in Table 2. We use Fides to ensure that the untrusted **helper** (**CP5**) compartment can neither access the votes table nor escalate to election official privilege. To this end, the security engineer defines the compartment access policy, represented as an access matrix in Figure 1, that does not allow **helper** (**CP5**) to call functions in any other compartment except the OCaml standard library in **CP6**. During runtime, the hardware monitors the control flow, and traps when the control flow does not comply with the defined access policy.

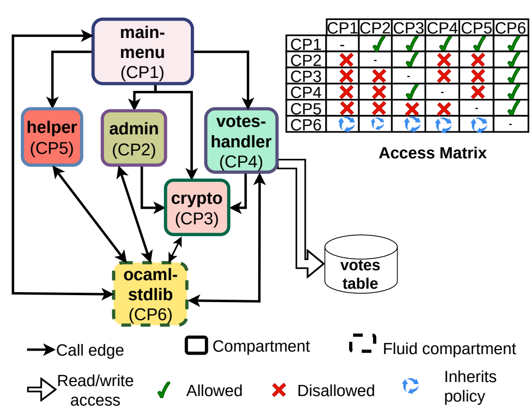

Fig. 1. EVM application: The edges between the compartments depict the permitted control flow between compartments. Only **CP4** compartment has access to the votes table. The allowed compartment transitions are depicted in the access matrix.

3.1.1 *Addressing challenge C1.*

Recall that Fides does not have an explicit data compartment but relies on the memory safety of the OCaml language and hardware-accelerated fat pointers for C code. In the EVM, the pointer to the `votes table` is defined with a local scope of **CP4**. To access the votes table, the attacker either has to exploit some memory vulnerability or should be able to invoke a function that has access to the votes table. The former is prevented by memory safety, and the latter by preventing the untrusted **CP5** from accessing **CP4**, either directly or transitively, with the help of the compartment access policy.

3.1.2 Addressing challenge C2. Defining code and data compartment policies for higher-order function closures is tricky. For example, consider the following code:

```
let res_tab = Array.make num_candidates 0 (* candidate_id->num_votes *)
let count_votes votes_arr (* decrypted votes array *) =
  let inc_vote candidate_id =
    res_tab.(candidate_id) <- res_tab.(candidate_id) + 1
  in
  Array.iter inc_vote votes_arr
```

Listing 1. A partial listing of vote_counting.ml from EVM application

The code above belongs to the vote counting module in compartment **CP4**. The array res_tab is a table that contains the results of the election, mapping candidate IDs to the number of votes cast for this candidate. The counts are initialized to 0. Observe that the higher-order function inc_vote belongs to **CP4**, closes over data (res_tab) that belongs to **CP4**, but is invoked for every array element by Array.iter. The question is, which compartment should Array.iter be placed in so that (1) the security guarantees are preserved and (2) the execution is efficient. There are three options: (a) duplicate Array.iter in each compartment where it is used, (b) place Array.iter in the same compartment as inc_vote, i.e. **CP4**, or (c) place Array.iter in the **CP6** compartment (with the rest of the OCaml standard library functions) and allow **CP6** to access **CP4**, by enabling it in the access matrix.

Standard library functions like Array.iter are pervasively used throughout the application. Duplicating them in every compartment is simple but inefficient (due to larger binary sizes and thus lowering instruction cache efficiency). Placing the Array.iter in **CP4** is insecure since every compartment that needs to access Array.iter will gain access to all of **CP4**. Placing Array.iter in a separate compartment, say **CP6**, may seem like the better choice in terms of security, but it is inefficient since every iteration of the array will need to switch from Array.iter's compartment **CP6** to inc_vote's compartment **CP4**. In addition, this scheme also opens up security issues since the compartment **CP6** is allowed access to **CP4**. All other compartments needing access to Array.iter should be allowed to access **CP6**. An attacker can misuse this scheme to stage a *confused deputy attack* [19] as shown on the left of Figure 2. The attacker in the untrusted **CP5** compartment can call Array.iter with inc_vote and a maliciously crafted votes_arr with forged votes, thereby updating the results table res_tab. Thus, none of the three options can securely handle higher-order functions.

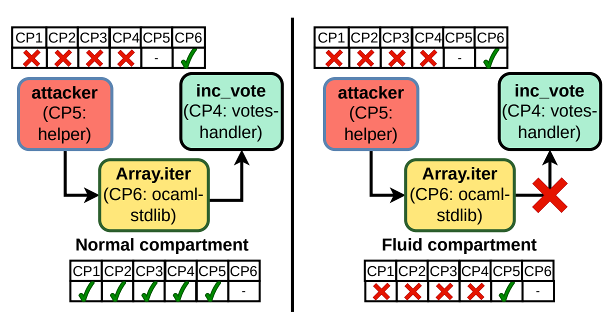

Fig. 2. **CP6** implemented as a normal compartment (Left) vs. a fluid compartment (right).

To securely and efficiently compartmentalize higher-order functions, Fides introduces the notion of *fluid compartments*. A fluid compartment does not have a fixed compartment policy of its own but inherits the compartment access policy of its caller compartment. The policy on the right of Figure 2 shows **CP6** marked as a fluid compartment. When the attacker invokes `Array.iter` with `inc_vote` and a malicious `votes_arr`, `Array.iter` inherits the **CP5**'s access policy. Since **CP5** is not allowed to access **CP4**, the call to `inc_vote` fails. This prevents confused deputy attacks.

As we will see in later sections, switching compartments requires saving and restoring compartment-local context. Our results in §6 show that the cost of switching to a fluid compartment is closer to an intra-compartment call and is much cheaper than an inter-compartment call.

3.1.3 Addressing challenge C3 with FIDES. Existing code compartment schemes only support typical call and return sequences. OCaml supports exceptions and tail-call optimisation, whose control flow is more complex than the typical function call and return sequence. Non-local control flow makes the implementation of code compartment schemes challenging.

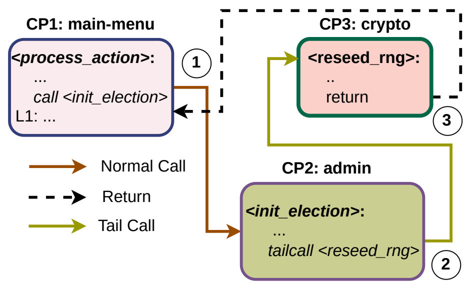

Fig. 3. FIDES support for tail-calls across compartments. After the tail-call from **CP2** to **CP3**, FIDES ensures that control returns to **CP1** skipping **CP2**

Consider the example given in Figure 3. Before the machine is used for an election, its state must be reset. The function `init_election` resets the machine, preparing for the new election. As part of the procedure, it reseeds the random number generator. The function `process_action` in the main menu compartment **CP1** calls `init_election` in **CP2**, which in turn tail calls the function `reseed_rng` which reseeds the random number generator. Without compartments, the function `reseed_rng` returns to `process_action`, skipping `init_election` in the return path. As we will see in §4, whenever we switch compartments, the hardware saves and restores compartment state in a separate *security monitor* (SM) stack that is inaccessible by the user code. In the presence of tail calls, the hardware compartment scheme must be made aware of the semantics of tail calls so that the SM stack is appropriately unwound. The same holds for exceptions, which will unwind the stack until the matching exception handler. FIDES extends the code compartment scheme to permit tail call optimisations and exceptions.

4 FIDES Formal Model

In this section, we present the design of FIDES with the help of formal operational semantics for a core language inspired by RISC-V. Using the model, we show how the security guarantees of the FIDES are preserved.

Notations. We use v^* to represent an array of v's indexed by an integer. We use $\overline{v}$ for a list of values v, $x :: xs$ to represent a list with a head x and a tail xs, $l@l'$ to represent a list obtained by appending lists l and l', and [] for the empty list. We define $\mathrm{trunc}(l, n)$ to be ls, where $l = lp@ls$ and $|ls| = n$. Intuitively, if the list l is used as a stack, $\mathrm{trunc}(l, n)$ pops the stack until the length of the stack is n. We use $\emptyset$ for an empty map.

4.1 Syntax

Figure 4 presents the syntax of our core language. The values v in our language are either word-sized integers or pointers w or fat pointers fp. Fat pointers are only used in C and consists of 4 word-sized fields – base α and bound β used for spatial safety, a cookie k for temporal safety, and

Word-sized integer w		
Fat pointer fp	::=	$\langle \alpha, \beta, \kappa, p \rangle$
Base α	::=	w
Bound β	::=	w
Cookie κ	::=	w
Pointer p	::=	w
Values v	::=	$w \mid fp$
compartment ID cid	::=	$\mathbb{N}$

(a) Values

Integer registers r	::=	$r_0 \dots r_{31}$
Fat pointer register fr		
Register R	::=	$r_i \mid fr$
Instructions i	::=	call r | tailcall r | return
	|	extcall r | callback r
	|	raise | pushtrap r | poptrap
	|	val r fr | load R r | store r R
	|	fatmalloc fr r | fatfree fr
	|	checkcap cid | push R | pop R

(b) Registers and Instructions

Fig. 4. Syntax

Program P	::=	i^*
Program counter pc	::=	w
Stack σ	::=	$\overline{v}$
Heap H	::=	w^*
Register map ρ	::=	$\emptyset \mid \rho[R \mapsto v]$
Access matrix $\mathcal{A}$	::=	$cid \times cid$
Current compartment ID ϕ	::=	cid
Fluid compartment ϕ_f	::=	cid
Compartment Map C	::=	$\emptyset \mid C[cid \mapsto (pc, pc)]$
Security monitor (SM) $\mathcal{S}$	::=	$\{\phi, \sigma_{\mathcal{S}}, C, \mathcal{A}\}$
SM stack $\sigma_{\mathcal{S}}$	::=	$\overline{(pc, cid)}$
Length of program stack l_{st}	::=	w
Length of SM stack l_{smst}	::=	w
Exception stack σ_{exn}	::=	$\overline{(pc, \phi, l_{st}, l_{smst})}$
Program State Π	::=	$\lceil pc, H, \sigma, \rho, \sigma_{exn} \rceil$
Machine State $\sum$	::=	$\lVert P, \mathcal{S}, \Pi \rVert$

Fig. 5. Runtime structures

the memory pointer p. The compartment ID, cid, is natural number used to uniquely identify a compartment.

The machine has 32 integer registers r_i and a single fat pointer register fr. We assume that the FIDES C and OCaml compilers target the instruction set i. The language includes instructions for indirect call and tailcall, and return. The instructions extcall and callback are part of the OCaml–C foreign function interface (FFI) to invoke C code from OCaml and vice versa, respectively. The instructions raise, pushtrap and poptrap are used for exceptions while load and store are used to access memory. The data compartments in C are enforced with the help of the val instruction, which validates the data access. The instructions fatmalloc and fatfree are used to allocate and free memory in C. Instructions push and pop respectively push and pop values from the program stack, while checkcap enforces code compartments. We assume but do not model the standard arithmetic and logical instructions.

4.2 Runtime structures

Figure 5 describes the runtime state of the machine. The program P is an array of instructions indexed by a word-sized program counter pc. The program stack σ is a list of values, and the heap H is an array of words. The stack is primarily used for spilling registers. All the objects are allocated on the heap. The register map ρ maps the registers to values.

The machine state Σ is a triple with the program P, the security monitor (SM) state $\mathcal{S}$ and the program state Π. The program state consists of those parts of the state that can be directly manipulated by the instructions. This includes the program counter pc, the heap H, the stack σ, the register map ρ and the exception stack σ_{exn}. The SM $\mathcal{S}$, which cannot be directly accessed by user code, consists of the current compartment ID ϕ, the SM stack $\sigma_{\mathcal{S}}$, the compartment map C and the access matrix $\mathcal{A}$. The compartment map C maps the compartment IDs to non-overlapping ranges of program counters. The access matrix $\mathcal{A}$ and the compartment map C are defined by the expert security engineer separately from the program and do not change during the execution of the program.

Fides' compiler assembles code such that functions mapped to the same compartment are placed adjacent to each other in the program. This lets us quickly check whether a pc belongs to a given compartment by performing range checks. The access matrix $\mathcal{A}$ is a binary relation on compartment IDs. If $(c_1, c_2) \in \mathcal{A}$, then a function in c_1 is allowed to call any function in c_2. We assume that there is a distinguished compartment ID ϕ_f for the fluid compartment. We defer the details of the exception stack σ_{exn} and SM stack σ_S to the rules that manipulate them.

4.3 Calling convention

Our formal model uses a simple calling convention inspired by RISC-V ABI, which is extended to support fat pointers. We assume that every function takes at most one integer and one pointer. The arguments are passed in registers. Given that fat pointers cannot be stored in an integer register, different language combinations use different argument registers. Table 3 presents the argument registers used in different combinations of languages.

When calling between C functions, the integer argument is passed in r_1 and the fat pointer in fr. OCaml does not use fat pointers, calls between OCaml functions uses $r1$ for integer argument and r_2 for the pointer. When calling C from OCaml, the pointer in r_2 is promoted to a fat pointer and passed in fr to the callee. Similarly, when calling from C to OCaml, the fat pointer in fr is demoted to a word-sized pointer and passed in r_2.

Table 3. Argument registers used in function calls.

Caller	**Callee**	r_1	r_2	fr
C	C	int	–	fat pointer
OCaml	OCaml	int	pointer	–
OCaml	C	int	pointer	fat pointer(r_2)
C	OCaml	int	pointer(fr)	fat pointer

Like OCaml, all registers are caller-saved registers. The registers $r_3 \ldots r_{31}$ are temporaries. The set of temporary registers is indicated by R_{tmp}. During a call, the return address is saved in the r_0 register. For return values, OCaml returns both integer and pointer results in r_1, whereas C returns integers in r_1 and fat pointers in fr.

Note that these simplifications to the calling convention have only been made to the formal model. Fides implementation does not modify the ABI of the C and OCaml programming languages and is extended to support the compartment scheme. We defer the details of this to later sections.

4.4 Operational Semantics

In this section, we present a small-step operational semantics for our core language. Every reduction step is of the form $\Sigma \longrightarrow \Sigma'$ where the machine takes one step from state Σ to Σ'.

4.4.1 **Call and return instructions.** Figure 6 shows the semantics of call and return instructions. The rules with the prefix Comp in their names capture the semantics of inter-compartment control-flow transition whereas the other rules correspond to intra-compartment transitions. We use an auxiliary definition InCompto check whether a given pc value lies within a compartment boundary.

Definition 4.1 (Intra-compartment check). The intra-compartment check InComp is defined as $\textsc{InComp}(C, \phi, pc) = (pc >= \mathsf{fst}(C[\phi]) \land pc <= \mathsf{snd}(C[\phi])) \lor (pc >= \mathsf{fst}(C[\phi_f]) \land pc <= \mathsf{snd}(C[\phi_f]))$

Intuitively, $\textsc{InComp}(C, \phi, pc)$ holds when pc is either within the current compartment ϕ or is within the fluid compartment ϕ_f.

$$\frac{\boxed{\textsc{Call}}\quad P[pc] = \mathsf{call}\ r \qquad pc' = \rho[r] \qquad \textsc{InComp}(\mathcal{S}.C, \mathcal{S}.\phi, pc')}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc', H, \sigma, \rho[r_0 \mapsto pc+1], \sigma_{exn}\rceil\|}$$

$$\frac{\boxed{\textsc{TailCall}}\quad P[pc] = \mathsf{tailcall}\ r \qquad pc' = \rho[r] \qquad \textsc{InComp}(\mathcal{S}.C, \mathcal{S}.\phi, pc')}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc', H, \sigma, \rho, \sigma_{exn}\rceil\|}$$

$$\frac{\boxed{\textsc{Return}}\quad P[pc] = \mathsf{return} \qquad pc_{ret} = \rho[r_0] \qquad \textsc{InComp}(\mathcal{S}.C, \mathcal{S}.\phi, pc_{ret})}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc_{ret}, H, \sigma, \rho, \sigma_{exn}\rceil\|}$$

$$\frac{\boxed{\textsc{CompCall}}\quad P[pc] = \mathsf{call}\ r \qquad pc' = \rho[r] \qquad \neg\,\textsc{InComp}(C, \phi, pc') \qquad P[pc'] = \mathsf{checkcap}\ \phi' \qquad (\phi, \phi') \in \mathcal{A}}{\begin{array}{c}\|P, \{\phi, \sigma_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \\ \|P, \{\phi', (pc+1, \phi) :: \sigma_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc', H, \sigma, \rho[r_0 \mapsto |P|][R_{tmp} \mapsto 0], \sigma_{exn}\rceil\|\end{array}}$$

$$\frac{\boxed{\textsc{Checkcap}}\quad P[pc] = \mathsf{checkcap}\ \phi}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc+1, H, \sigma, \rho, \sigma_{exn}\rceil\|}$$

$$\frac{\boxed{\textsc{CompReturn}}\quad P[pc] = \mathsf{return} \qquad \rho[r_0] = |P| \qquad (pc_{ret}, \phi') :: \sigma'_{\mathcal{S}} = \sigma_{\mathcal{S}}}{\|P, \{\phi, \sigma_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \{\phi', \sigma'_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc_{ret}, H, \sigma, \rho[R_{tmp} \mapsto 0], \sigma_{exn}\rceil\|}$$

$$\frac{\boxed{\textsc{CompTailCall1}}\quad \begin{array}{c} P[pc] = \mathsf{tailcall}\ r \\ \rho[r_0] \neq |P| \qquad pc' = \rho[r] \qquad \neg\,\textsc{InComp}(C, \phi, pc') \qquad P[pc'] = \mathsf{checkcap}\ \phi' \qquad (\phi, \phi') \in \mathcal{A}\end{array}}{\begin{array}{c}\|P, \{\phi, \sigma_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \\ \|P, \{\phi', (\rho[r_0], \phi) :: \sigma_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc', H, \sigma, \rho[r_0 \mapsto |P|][R_{tmp} \mapsto 0], \sigma_{exn}\rceil\|\end{array}}$$

$$\frac{\boxed{\textsc{CompTailCall2}}\quad \begin{array}{c} P[pc] = \mathsf{tailcall}\ r \\ \rho[r_0] = |P| \qquad pc' = \rho[r] \qquad \neg\,\textsc{InComp}(C, \phi, pc') \qquad P[pc'] = \mathsf{checkcap}\ \phi' \qquad (\phi, \phi') \in \mathcal{A}\end{array}}{\|P, \{\phi, \sigma_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \{\phi', \sigma_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc', H, \sigma, \rho[R_{tmp} \mapsto 0], \sigma_{exn}\rceil\|}$$

Fig. 6. Semantics of calls and returns.

The rule $\boxed{\textsc{Call}}$ shows the semantics of the call instruction for an intra-compartment call. The target program counter pc' is in the current compartment or belongs to the fluid compartment. If so, the program counter is updated to pc' and the return address register r_0 is set to $pc + 1$. As an aside, note that since all registers are caller-saved in the formal model, the compiler saves the return address register r_0 on the stack at the entry to a function and restores it before returning using the push and pop instructions. Intra-compartment tail call (rule $\boxed{\textsc{TailCall}}$) is similar to call, but it does not modify the return address register r_0. On an intra-compartment return (rule $\boxed{\textsc{Return}}$), we check that the return address is indeed in the current compartment or the fluid compartment. Then the program counter is updated to the address in the r_0 register.

The rule $\boxed{\textsc{CompCall}}$ captures the semantics of inter-compartment call – the target program counter pc' is not in the current compartment ϕ or in the fluid compartment ϕ_f. We perform a couple of integrity checks to see whether this inter-compartment call is allowed. First, we check whether the instruction at the target program counter pc' is a checkcap ϕ' instruction, where ϕ' is the target compartment ID. Fides compiler inserts the checkcap instruction with the corresponding compartment ID at every compartment entry point. The instruction checkcap serves the same purpose as *endbr* instruction in Intel's Control-flow Enforcement Technology (CET) [21], to protect against attacks such as return-oriented programming (ROP) and jump-oriented programming (JOP). If *endbr* is not found at the target of an indirect jump or call, then the processor traps, thwarting the attempted control-flow hijack. We model a similar behaviour by expecting checkcap instruction at the target of an inter-compartment call. Note that the checkcap instruction itself is a no-op (rule $\boxed{\textsc{Checkcap}}$). Finally, we check that the compartment transition is permitted by the access matrix $(\phi, \phi') \in \mathcal{A}$.

When the integrity checks hold, we transfer control to the target function in the new compartment. We update the compartment ID in the SM state $\mathcal{S}$ to the target compartment ϕ'. We push the return address and the current compartment ID to the SM stack. The program counter is updated to the target pc'. The return address register r_0 is set to a special program counter value $|P|$ outside the range of the program. Note that the program P is an array of instructions indexed by the program counter and hence the index $|P|$ lies outside of the program P. This special return address is used to identify whether the control returns to another compartment on the return path. Saving the return address on the SM's private stack prevents an attacker-controlled callee compartment from returning to any arbitrary instruction in the caller's compartment. At most, an attacker can subvert control flow to some location within the current or fluid compartment but cannot enter other compartments. Finally, we also zero out all of the temporary registers $R_{tmp} = r_3 \ldots r_{31}$ to avoid leaking any information from the caller to the callee compartment.

The rule $\boxed{\textsc{CompReturn}}$ models inter-compartment return, which is identified by the return address in r_0 being $|P|$. In this case, the SM stack is popped to get the previous compartment ID ϕ' and the return address pc_{ret}. The SM and program state are updated to the caller information. We also zero out the temporary registers to avoid leaking information across compartments.

The interaction of tail calls and compartments are interesting. Let us use the notation $f : \phi$ to indicate that the function f belongs to the compartment ϕ. The rule $\boxed{\textsc{CompTailCall1}}$ specifies the semantics of an inter-compartment tail call preceded by an intra-compartment call. For example, consider the calling sequence below:

$$f : \phi \xrightarrow{call} g : \phi' \xrightarrow{call} h : \phi' \xrightarrow{tailcall} i : \phi''$$

The rule specifies the behaviour of $h : \phi'$ calling $i : \phi''$. Importantly, due to the tail call, i must return to g. In the premise, we identify that the tail call has been preceded by an intra-compartment call since $\rho[r_0] \neq |P|$. The rest of the premises are the same as the inter-compartment call (rule

$$\frac{\boxed{\textsc{PushTrap}} \quad P[pc] = \text{pushtrap } r \qquad pc_{exn} = \rho[r] \qquad \textsc{InComp}(\mathcal{S}.C, \mathcal{S}.\phi, pc_{exn})}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn} \rceil\| \quad \to \quad \|P, \mathcal{S}, \lceil pc+1, H, \sigma, \rho, (pc_{exn}, \mathcal{S}.\phi, |\sigma|, \mathcal{S}.|\sigma_{sm}|) :: \sigma_{exn} \rceil\|}$$

$$\frac{\boxed{\textsc{PopTrap}} \quad P[pc] = \text{poptrap} \qquad \_ :: \sigma'_{exn} = \sigma_{exn}}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn} \rceil\| \quad \to \quad \|P, \mathcal{S}, \lceil pc+1, H, \sigma, \rho, \sigma'_{exn} \rceil\|}$$

$$\frac{\boxed{\textsc{Raise}} \quad P[pc] = \text{raise} \qquad (pc_{exn}, \phi', l_{st}, l_{smst}) :: \sigma'_{exn} = \sigma_{exn} \qquad \sigma' = \text{trunc}(\sigma, l_{st}) \qquad \sigma'_{sm} = \text{trunc}(\sigma_{sm}, l_{smst})}{\|P, \{\phi, \sigma_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc, H, \sigma, \rho, \sigma_{exn} \rceil\| \quad \to \quad \|P, \{\phi', \sigma'_{\mathcal{S}}, C, \mathcal{A}\}, \lceil pc_{exn}, H, \sigma', \rho[R_{tmp} \mapsto 0], \sigma'_{exn} \rceil\|}$$

Fig. 7. Semantics of the exceptions

$\boxed{\textsc{CompCall}}$). The only difference in the conclusion compared to the rule $\boxed{\textsc{CompCall}}$ is that rather than $pc + 1$ being pushed onto the SM stack, since the call is a tail call, we push the return address of the caller $\rho[r_0]$. This ensures that the control return to the caller of the current function when the callee returns.

The rule $\boxed{\textsc{CompTailCall2}}$ specifies the semantics of an inter-compartment tail call preceded by an inter-compartment call, such as the calling sequence presented below:

$$f : \phi \xrightarrow{call} g : \phi' \xrightarrow{tailcall} i : \phi''$$

The rule specifies the behaviour of $g : \phi'$ calling $i : \phi''$. Here, $i : \phi''$ must return to $f : \phi$, skipping the function g and the compartment ϕ'. We know that the current function was entered through an inter-compartment call since the return address r_0 is $|P|$. The rest of the premises are the same as rule $\boxed{\textsc{CompTailCall1}}$. Importantly, in this case, we do not push an entry to the SM stack compared to $\boxed{\textsc{CompTailCall1}}$, which allows this compartment to be skipped on the return path. The rest of the conclusion is the same as $\boxed{\textsc{CompTailCall1}}$.

4.4.2 **Exceptions.** Similar to tail calls, OCaml exceptions also have interesting interactions with code compartments. When an exception is raised, in addition to unwinding the program stack, the machine also needs to unwind the SM stack. In the formal model, unlike OCaml, we assume that there is a unique, unnamed exception. Hence, raise does not take any parameter and unwinds the control to the closest matching handler. OCaml exception handlers are compiled to pushtrap and poptrap instructions, which delimit the exception handler scope. Let $[\![O]\!]$ represent the compilation of the OCaml program O to P. Then $[\![$ `try e with E ->...` $]\!]$ is defined as pushtrap r; $[\![e]\!]$; poptrap where r holds the program counter corresponding to the exception handler code $[\![$ `E ->...` $]\!]$.

The rule $\boxed{\textsc{PushTrap}}$ shows the semantics of the push instruction. pushtrap r takes the register r that holds the program counter of the exception handler pc_{exn} as an argument. We check that the exception handler program counter pc_{exn} is indeed within the current compartment. If so, a new entry is pushed onto the exception handler stack σ_{exn}, with the exception handler program counter pc_{exn}, the current compartment ID $\mathcal{S}.\phi$, and the lengths of the current program and SM stack. The latter two are used during raise to unwind the corresponding stacks.

The rule $\boxed{\textsc{PopTrap}}$ simply pops the exception handler stack, thus removing the exception handler from the scope. When an exception is raised (rule $\boxed{\textsc{Raise}}$), we find the most recent exception handler information from the exception stack and truncate both the program stack and the SM stack to the length that they were at the point of installing the exception handler. Recall that $\mathsf{trunc}(\sigma, n)$ pops elements from the stack σ until the length of the stack is n. Observe that raise permits throwing exceptions within and across the compartments. In the case of a cross-compartment exception, we had already validated that pc_{exn} is within the compartment ϕ' when the exception handler was installed (in pushtrap). On a raise, we also unconditionally reset all the temporary registers to 0 to prevent the possibility of information leaking across compartments through temporaries.

4.4.3 **Memory access.** Figure 8 presents the semantics of the memory access instructions. Unlike code compartments, Fides does not have explicit data compartments. Instead, it provides the guarantee that a function can only access the data that is transitively accessible from its locals, arguments and global data. This makes it easy to use higher-order functions which may close over data allocated in other compartments without worrying about the data compartments where the environment variables belong to. While OCaml ensures memory safety at the language level, C does not. Hence, we utilise a hardware-based fat pointer scheme, Shakti-MS [11], to ensure memory safety in the untrusted C dependencies. We extend the fat pointer scheme of Shakti-MS to accommodate mixed safe-and-unsafe language applications.

Rule $\boxed{\textsc{FatMalloc}}$ presents the semantics of an allocation in C. The instruction fatmalloc fr r takes the size of memory to allocate in words, and if successful, the resultant fat pointer is stored in the fr register. We assume a primitive malloc instruction that takes the size in words and returns a pointer to this allocated memory. Every allocation in the C heap includes a header that has a randomized cookie value for temporal safety. Note that in the antecedent, we request malloc to allocate a memory region 1 word larger than the original request. We also assume a primitive function rand than returns a random word, using which we obtain a fresh cookie value k. The pointer returned p points to the word after the header. We update the heap such that the header word is set to the fresh cookie value and the rest of the fields in the newly allocated region are zero'ed out. In the conclusion of the rule, we update the fr register with the newly crafted fat pointer.

Rule $\boxed{\textsc{Val}}$ presents the semantics of the val r fr instruction which validates a fat pointer fr, and if successful, returns the pointer value in r. The checks include the spatial checks to see whether the pointer is within the base and the bound, and the temporal check to see whether the cookie k matches the cookie in the header of the memory region. We also permit the cookie in the fat pointer to be 0. This allows C code to access objects on the OCaml heap. A fat pointer with zero cookie is only crafted during an external call when pointers to OCaml objects are shared with C. We defer the details of this to the semantics of the extcall instruction. If the validation is successful, the register r is updated to have the pointer value p. Before loading from, storing to, freeing and passing a fat pointer from C to OCaml, the compiler inserts val instruction to check its validity.

Rule $\boxed{\textsc{FatFree}}$ describes the semantics of freeing memory in C. The fatfree instruction uses the primitive free instruction to return memory back to the operating system. The header of the freed memory region is set to a fresh cookie value to prevent use-after-free issues. As mentioned earlier, since fatfree was preceded by the val instruction, double-free issues are impossible as the cookie validation will fail for the second fatfree. Note that the OCaml allocator and the garbage collector directly utilise the primitive malloc and free.

Rule $\boxed{\textsc{LoadW}}$ describes the semantics of load r_d r_a where the address is in r_a and the result will be stored in r_d. The rule is straightforward and updates the destination register with the value from

$$\boxed{\textsc{FatMalloc}}$$
$$\frac{P[pc] = \mathsf{fatmalloc}\ fr\ r \qquad sz = \rho[r] \qquad \alpha = \mathsf{malloc}(sz+1) \\ \kappa = \mathsf{rand}() \qquad p = \alpha + 1 \qquad \beta = \alpha + sz + 1 \qquad H' = H[\alpha \mapsto \kappa][(\alpha+1)\dots(\alpha+sz) \mapsto 0]}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc+1, H', \sigma, \rho[fr \mapsto \langle \alpha, \beta, \kappa, p\rangle], \sigma_{exn}\rceil\|}$$

$$\boxed{\textsc{Val}}$$
$$\frac{P[pc] = \mathsf{val}\ r\ fr \qquad \langle \alpha, \beta, \kappa, p\rangle = \rho[fr] \qquad p > \alpha \qquad p < \beta \qquad (H[\alpha] = \kappa \vee \kappa = 0)}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc+1, H, \sigma, \rho[r \mapsto p], \sigma_{exn}\rceil\|}$$

$$\boxed{\textsc{FatFree}}$$
$$\frac{P[pc] = \mathsf{fatfree}\ fr \qquad \langle \alpha, \beta, \kappa, p\rangle = \rho[fr] \qquad \mathsf{free}(\alpha) \qquad \kappa' = \mathsf{rand}()}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc+1, H[\alpha \mapsto \kappa'], \sigma, \rho, \sigma_{exn}\rceil\|}$$

$$\boxed{\textsc{LoadW}}$$
$$\frac{P[pc] = \mathsf{load}\ r_d\ r_a \qquad a = \rho[r_a]}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc+1, H, \sigma, \rho[r_d \mapsto H[a]], \sigma_{exn}\rceil\|}$$

$$\boxed{\textsc{LoadFP}}$$
$$\frac{P[pc] = \mathsf{load}\ fr\ r \qquad a = \rho[r]}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \\ \|P, \mathcal{S}, \lceil pc+1, H, \sigma, \rho[fr \mapsto \langle H[a], H[a+1], H[a+2], H[a+3]\rangle], \sigma_{exn}\rceil\|}$$

$$\boxed{\textsc{StoreW}}$$
$$\frac{P[pc] = \mathsf{store}\ r_a\ r_v \qquad a = \rho[r_a]}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc+1, H[a \mapsto \rho[r_v]], \sigma, \rho, \sigma_{exn}\rceil\|}$$

$$\boxed{\textsc{StoreFP}}$$
$$\frac{P[pc] = \mathsf{store}\ r\ fr \qquad \langle \alpha, \beta, \kappa, p\rangle = \rho[fr] \qquad a = \rho[r]}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \\ \|P, \mathcal{S}, \lceil pc+1, H[a \mapsto \alpha][a+1 \mapsto \beta][a+1 \mapsto \kappa][a+3 \mapsto p], \sigma, \rho, \sigma_{exn}\rceil\|}$$

Fig. 8. Semantics of memory access intructions

the heap address. Note that this instruction is used by OCaml for loading integers and pointers, but C only uses them to load integers. For (fat) pointers in C, rule $\boxed{\textsc{LoadFP}}$ applies. Here the destination register is the fat pointer register fr. load fr r loads 4 consecutive words from the memory address pointed to by r and loads that in fr. The rules $\boxed{\textsc{StoreW}}$ and $\boxed{\textsc{StoreFP}}$ are duals of the load instruction.

It is useful to see how these instructions are used by the C compiler. Consider the following C code: `intptr_t *p; ...; v = *p`. For `*p`, the Fides C compiler generates val r_x fr; load r_y r_x. The fat pointer corresponding to p will be in the fr register. Using the val instruction, we extract the

$$\boxed{\textsc{ExtCall}}$$

$$\frac{P[pc] = \text{extcall } r \qquad p = \rho[r_2] \qquad l_{obj} = \text{objlen}(p) \qquad \|P[pc \mapsto \text{call } r], \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \rightarrow \|P', \mathcal{S}', \lceil pc', H, \sigma, \rho', \sigma_{exn}\rceil\|}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}', \lceil pc', H, \sigma, \rho'[fr \mapsto \langle p-1, p+l_{obj}, 0, p\rangle][r_2 \mapsto 0], \sigma_{exn}\rceil\|}$$

$$\boxed{\textsc{Callback}}$$

$$\frac{P[pc] = \text{callback } r \qquad \langle \alpha, \beta, \kappa, p\rangle = \rho[fr] \qquad \kappa = 0 \qquad \|P[pc \mapsto \text{call } r], \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \rightarrow \|P', \mathcal{S}', \lceil pc', H, \sigma, \rho', \sigma_{exn}\rceil\|}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn}\rceil\| \quad \rightarrow \quad \|P, \mathcal{S}', \lceil pc', H, \sigma, \rho'[r_2 \mapsto p][fr \mapsto \langle 0,0,0,0\rangle], \sigma_{exn}\rceil\|}$$

Fig. 9. Semantics of the foreign function interface.

pointer in r_x register. Since the data pointed by r_x to is an integer, the destination register r_y in the load is an integer register. Now consider the following C code: `intptr_t **p; ...; v = *p`. For `*p`, the Fides C compiler generates val r_x fr; load fr r_x. Unlike the previous example, the data pointed to by r_x is a fat pointer. Hence, the destination of the load is the fat pointer register fr. As shown in rule $\boxed{\textsc{LoadFP}}$, this load instruction loads 4 consecutive words from the address pointed to by r_x.

4.4.4 **Foreign function interface.** The rules in Figure 9 describe the semantics of the foreign function interface between OCaml and C. They broadly behave similarly to the call instruction. In fact, we use the reduction step for the call instruction to describe the semantics of foreign function calls. The main challenge here is the translation of pointer arguments when passed from OCaml to C and vice versa.

Rule $\boxed{\textsc{ExtCall}}$ describes the semantics of extcall instruction that calls a C function from OCaml. As described in §4.3, OCaml functions pass the integers in $r1$ and pointers in $r2$. The pointer in $r2$ must be translated to a fat pointer to pass to the C function. We can do this thanks to the fact that OCaml object headers encode the object length. We assume a primitive objlen function that returns the object length. Using the pointer and the object length, we craft a fat pointer with the cookie value 0 as described in §4.4.3.

Rule $\boxed{\textsc{Callback}}$ describes the semantics of callback instruction that calls an OCaml function from C. In this case, on the caller (C) side, r_1 holds the integer argument, and fr holds the pointer argument. OCaml can only work with memory allocated in the OCaml heap and not the C heap. Hence, the only valid pointer that can be passed from C to OCaml is a pointer to an object in the OCaml heap. Fat pointers to OCaml objects will have 0 for the cookie field. As before, the reduction step uses the reduction step for call instruction and then updates r_2 to the pointer value p and fr to the NULL fat pointer.

4.4.5 **Semantics of stack manipulation.** Figure 10 presents the semantics of push and pop instructions, whose semantics is straightforward.

4.5 Safety

In this section, we show how the safety guarantees of the compartment scheme are preserved in the operational semantics. Let us start with a few definitions.

Definition 4.2 (Well-formed compartment map). Given a program P and a compartment map C, we say that the compartment map is well-formed if the following conditions hold:

$$\boxed{\text{PUSH}}\quad \frac{P[pc] = \text{push } R}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn} \rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc + 1, H, \rho[R] :: \sigma, \rho, \sigma_{exn} \rceil\|}$$

$$\boxed{\text{POP}}\quad \frac{P[pc] = \text{pop } R \qquad v :: \sigma' = \sigma}{\|P, \mathcal{S}, \lceil pc, H, \sigma, \rho, \sigma_{exn} \rceil\| \quad \rightarrow \quad \|P, \mathcal{S}, \lceil pc + 1, H, \sigma', \rho[R \mapsto v], \sigma_{exn} \rceil\|}$$

Fig. 10. Semantics of push and pop instructions.

- $\forall (s, e) \in \text{range}(C), 0 <= s <= e < |P|$. The compartment ranges are bounded by the program size, and the end of the compartment range does not precede the start.
- $\forall (s, e), (s', e') \in \text{range}(C), s < s' \implies e < s'$. That is, the compartment ranges are non-overlapping.
- $\forall (s, e) \in \text{range}(C), P[e] = \text{return}$. The last instruction in a compartment is a return instruction. Hence, functions do not span multiple compartments.

Definition 4.3 (Safe machine state). Given a machine state $\Sigma = \|P, \mathcal{S}, \Pi\|$, we say that the machine state is safe (written safe(Σ)) if $\text{INCOMP}(\mathcal{S}.C, \mathcal{S}.\phi, \Pi.pc)$ holds.

Intuitively, we say that a machine is safe if the pc is within the current compartment boundary.

Definition 4.4 (Initial machine state). Given a program P, a well-formed compartment info C, and access matrix $\mathcal{A}$, the starting program counter pc, the initial compartment ϕ, the initial machine state is defined as follows $\Sigma_0 = \|P, \{\phi, [], C, \mathcal{A}\}, \lceil pc, \emptyset, [], \emptyset, \emptyset \rceil\|$.

THEOREM 4.5 (COMPARTMENT SAFETY). *Given a safe initial machine state Σ_0, where safe(Σ_0) holds, if $\Sigma_0 \longrightarrow^* \Sigma$, then safe($\Sigma$) holds.*

Proof sketch. The proof is by induction on the length of the trace. The base case is safe by definition. For the inductive case, the interesting instructions are those that change compartments.

- Inter-compartment call and tail call instructions (Figure 6) utilise the target compartment information ϕ', which is inserted by the compiler and known to be safe.
- For returning across compartments (rule $\boxed{\text{COMPRETURN}}$ in Figure 6), we need to show that pc_{ret} is in ϕ'. The return address pc_{ret} was pushed onto the SM stack by the inter-compartment call (rule $\boxed{\text{COMPCALL}}$). In rule $\boxed{\text{COMPCALL}}$, we know pc+1 also belongs to the current compartment ϕ because call cannot be the last instruction; since the compartment map C is well-formed, the last instruction in the compartment is a return instruction.
- Raising an exception can change compartments (rule $\boxed{\text{RAISE}}$ in Figure 7). But the target pc_{exn} is guaranteed to be in ϕ' since the validation was done in $\boxed{\text{PUSHTRAP}}$ when the exception stack entry was pushed into the exception stack □.

5 FIDES Implementation

We instantiate FIDES in the Shakti open-source RISC-V processor [14]. Currently, FIDES supports OCaml and C and is designed to run on Mirage unikernels[33]. MirageOS is a clean-slate unikernel containing a mix of OCaml and C code bases, making it suitable for evaluating FIDES on resource-constrained embedded systems. We extend the MirageOS backend to execute on baremetal RISC-V [23] processors. This section explains the changes made to the hardware and the software stack to support FIDES.

5.1 Hardware changes

5.1.1 **Code compartments.** Currently, Fides supports 256 compartments. We add one custom instruction, checkcap, to the RISC-V ISA. The processor expects this instruction to be present at all valid compartment entry points.

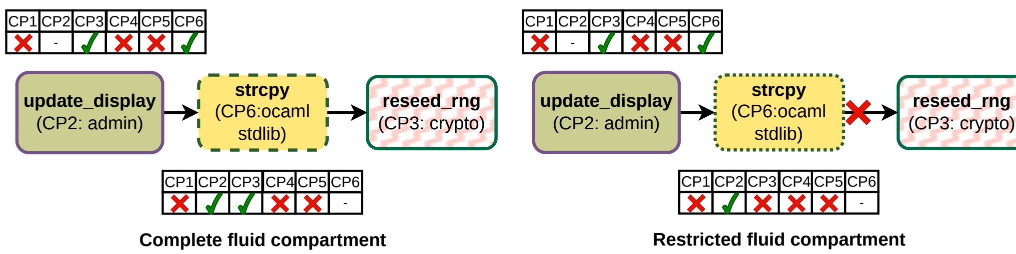

Fig. 11. Fluid compartment types

Restricted fluid compartment. As we have seen earlier (§3.1.2), Fides introduces a fluid compartment to compartmentalize commonly used higher-order functions in an efficient and secure way. There are other non-higher-order functions, such as string manipulation functions (`strcpy`, for example), which may presumably be placed in the fluid compartment for all the compartments to access. Since fluid compartment functions are given the same privileges as the caller compartment; they can access all the compartments that the caller compartment can access. However, this privilege is unnecessary; string manipulation functions are typically self-contained and do not need to call any other functions.

Worse, this can lead to exploits. Consider the case in the EVM application shown on the left of Figure 11. The function `update_display`, mapped to **admin** compartment (**CP2**), shows the ballot paper on the display. This invokes `strcpy` function to produce the output on the display. `strcpy` is mapped to **CP6** compartment. Since **CP6** is a fluid compartment, it can invoke **crypto** compartment (**CP3**) via the access policy it inherited from **CP2**. Any vulnerability present within `strcpy` function can be exploited by an attacker who can misuse this over-privilege to redirect control to the **crypto** compartment and manipulate sensitive cryptographic states maintained internally within the **crypto** compartment. By restricting the privilege of **CP6** further, we can eliminate this attack vector.

To this end, Fides also supports a *restricted* fluid compartment that can only call functions in the current compartment or the caller. The choice to allow calls to the functions in the caller compartment is to permit higher-order callbacks, which are pervasive in OCaml code. The compartment scheme presented in the formal model in §4 is termed as a *complete* fluid compartment. Observe that when **CP6** is marked as a restricted fluid compartment (RHS of Figure 11, `strcpy` can no longer call functions in the crypto compartment.

Compartment checks. The current compartment and the *pc* ranges of the two fluid compartments are maintained in RISC-V custom control and status registers (CSRs) [23] for fast access. All these CSRs are protected and can be accessed only by the SM.

We have modified the processor pipeline such that on every instruction, the *pc* is checked to see whether it is within the boundary of the current compartment or the fluid compartments. If not, the execution traps to the SM. At the SM, we check whether the access is allowed by the access matrix, and if allowed, performs the compartment context update necessary for switching compartments. Note that this differs from the formal model, which performs the checks only on control-flow instructions. By performing the checks on every instruction, our hardware provides stronger guarantees. For example, we do not need to assume that the compartments end with a return instruction; if it does not, the execution traps. Because of the pipelined nature of the check, checking on every instruction does not affect the processor's critical path and the processor's clock cycle count is not affected.

5.1.2 **Data compartments.** To support data compartments in C, hardware-assisted fat pointers [11] are used. The hardware and ISA are extended to support a new val instruction. The compiler inserts val instruction before dereferencing a fat pointer to enforce fine-grained spatial and temporal memory safety. Unlike the formal model, FIDES does not extend the hardware with a fat pointer register. Instead, use multiple integer registers to hold the fat pointer value in the register map. Our implementation, being targeted at small embedded systems, assumes a 32-bit address space but runs on 64-bit RISC-V hardware. This allows us to pack the 4 (32-bit) fields of the fat pointer into 2 (64-bit) integer registers. The instructions fatmalloc and fatfree, present in the formal model, are implemented as wrappers of malloc and free.

5.2 Software Changes

5.2.1 **Code compartments.** Maintaining a separate compartment mapping file is based on the industry-best standard approach of offloading the security-critical task of assigning compartments to a security engineer [9]. For every source code file in the application, FIDES expects a `.cap` file provided at compile time. For each function in a source code file, the corresponding `.cap` file contains the compartment ID for the function and a flag indicating whether the function is a valid entry point into the compartment. For example, in the code below,

```
<function>:<compartment ID>:<external>
count_votes : CP4 : ENTRY_POINT
inc_vote    : CP4 : NO_ENTRY_POINT
```

`inc_vote`, mapped to **CP4**, is not a compartment entry point and can only be used within **CP4** or the fluid compartments. The FIDES compiler also accepts a default compartment ID to which all the functions which have not been explicitly assigned a compartment ID is assigned to. FIDES OCaml and C compiler emits a checkcap instruction as the first instruction in each function that is tagged as a valid compartment entry point. A custom linker script is used to place all functions belonging to the same compartment in the same code section in the ELF generated. The SM derives the compartment boundary information at boot time by inspecting the ELF code sections. The SM code is placed in a reserved compartment.

The key task of the SM is to enforce the compartment access policy on all inter-compartment calls and returns. The SM executes with interrupts disabled, saves and restores compartment context on every compartment switch, and configures the CSRs on every compartment switch with appropriate compartment context. In the formal model, we build upon a calling convention, where it is assumed that all registers are caller-saved. However, the RISC-V C ABI includes callee-saved registers (`s0-s11`). On an inter-compartment C call, the caller does not trust the callee to not tamper with the callee-saved registers. Hence, the SM saves and restores callee-saved registers. OCaml does not follow the RISC-V ABI and does not have callee-saved registers. Hence, the SM does not save and restore registers on an inter-compartment OCaml call.

OCaml's garbage collection (GC) procedure scans the stack at the start of a GC to find the roots. OCaml uses the return address pushed onto the stack to identify GC roots in an activation frame. However, as seen in the formal model, FIDES changes the return address on an inter-compartment call to a known constant canary value. This interferes with stack scanning at the start of the GC. To get around this issue, FIDES uses a shadow stack into which a copy of the original returned address is pushed during an inter-compartment call. When the GC stack scanning procedure encounters a canary value, it consults the shadow stack to retrieve the original return address and continues scanning the rest of the stack.

5.2.2 **Data compartments.** To instrument C code with fat pointer checks, we modify the LLVM RISC-V backend to introduce a fat pointer transformation pass similar to Shakti-MS [11]. The fat pointer transformation pass (i) identifies all pointer allocations to stack, heap and global data regions and transforms them into fat pointers, (ii) inserts instructions at allocation points so that the fat pointer fields – base, bound and cookie – are populated, and (iii) inserts fat pointer validation val instruction before every fat pointer dereference.

In the formal model, we assume that all allocations happen on the heap. However, local allocations on the stack are standard in C. To ensure memory safety for stack allocations, we extend the fat pointer instrumentation to the program stack. For spatial safety, the base and bound of pointers into the stack are set to the base and bound of the stack frame. We admit that this is more coarse-grained than the object-level spatial safety that we have for heap allocations. With a pointer to a value on the stack memory location in that stack frame may be modified. However, this provides a good balance between performance and security. For temporal safety, each stack frame also includes a cookie value that is set to a fresh value when the function is entered and exited. This ensures that pointers into the stack are only valid when the frame that the pointer points to is currently active.

The OCaml runtime, implemented in C, is part of the trusted computing base (TCB) and is not compiled with the fat pointer instrumentation. The C memory safety instrumentation does not handle inline assembly automatically. We manually modify the inline assembly code to be aware of fat pointers and insert checkcap instructions at the function entry when necessary. Notably, the cost to do this is directly proportional to the size of the inline assembly code, which is expected to be small in real-world C libraries.

6 Results

6.1 Engineering effort

Fides extends LLVM version 11.1 and OCaml version 4.11.1 to add support for code compartments and memory safety in C. The changes to the OCaml compiler to support code compartments include 50 lines of code (LoC) in the RISC-V backend, 149 LoC in the frontend to handle `.cap` files, and 20 LoC in the GC to handle stack scanning using shadow stack. The changes to the LLVM backend and frontend to support code compartments are 155 and 37 LoC, respectively. The compartment description language allows a default compartment id to be specified for the functions in a given module. We have also extended the OCaml and C compiler and the `dune` and `ocamlbuild` build systems to support default compartment specification at the file and OCaml package level. The Shakti-MS fat pointer instrumentation pass in LLVM, which we build upon, consists of 2165 LoC. Observe that implementing Fides only requires minimal self-contained additions to the compilers.

Fides is realized on a Xilinx Artix-7 AC701 FPGA [57] with a default synthesis strategy. The baseline RISC-V core [14] consumes 36.0K look-up tables (LUTs) and 16.4K registers on the FPGA. The core with only support for fat pointers requires 36.3K LUTs and 16.5K registers, whereas the one with both fat pointers and code compartments requires 38.2K LUTs (+6.1%) and 17.4K LUTs (+6.0%). Importantly, the core's operating frequency is not affected by any of the modifications introduced by Fides.

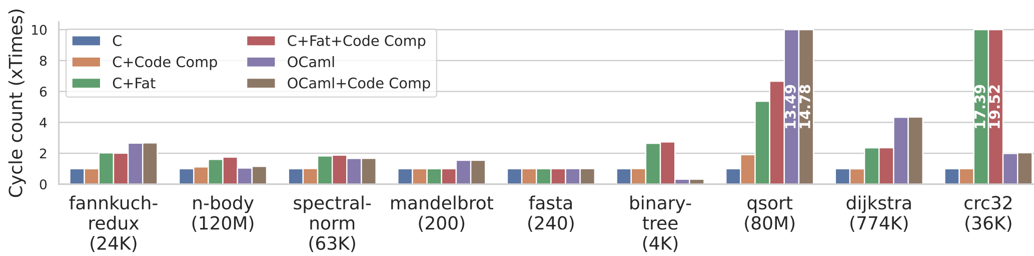

Fig. 12. Execution time (clock cycles) overhead w.r.t C baseline. The number of compartment transitions is specified in parentheses.

6.2 Microbenchmark

To quantify the overheads of compartmentalization of higher-order functions, we pick a simple program, `let f i v = arr.(i) <- v + 1 in Array.iteri f arr`, and evaluate 4 different compartmentalization schemes: (i) `f` and `Array.iteri` are in the same compartment (baseline), (ii) `Array.iteri` is placed in a restricted fluid compartment, (iii) `Array.iteri` is placed in a complete fluid compartment, and (iv) `f` and `Array.iteri` are placed in different compartments. The program is reminiscent of the example discussed in §3.1.2. The array `arr` has 100,000 elements in our benchmark run. We observe that placing `f` and `Array.iteri` in the same compartment takes 90M clock cycles. In the case where `Array.iteri` is placed in one of the fluid compartments, the program takes the same clock cycles as the baseline. This is because of the fact that the fluid compartment check is not in the critical path of the execution and does not affect the clock cycle. However, when `Array.iteri` in a different compartment, we see a 5.4× increase in the clock cycle count compared to the baseline. The overhead is high here since the work done by the higher-order function is far less than the overhead of saving and restoring the compartment context. Given that higher-order functions such as `Array.iteri` are pervasive in OCaml, fluid compartments prove to be essential to keep the performance overheads of code compartment scheme low. Moreover, as discussed in §3.1.2, fluid compartments also avoid the confused deputy problem when `Array.iteri` is placed in a different compartment.

6.3 Larger benchmarks

In this section, we quantify the following: (a) What is the cost of supporting code compartments? and (b) What is the cost of data compartments in OCaml compared to C hardened with fat pointers? Figure 12 shows the clock cycle overhead of enabling code and data compartments with respect to the C baseline. Columns marked with C and OCaml denote the baseline executables with both fat pointer and compartment checks disabled. All the microbenchmarks are taken from the computer language benchmarks game [50], except qsort, dijkstra and crc32, which were developed by us. mandelbrot and fasta are I/O intensive benchmarks, with mandelbrot containing negligible pointer operations.

6.3.1 **Cost of supporting code compartments.** We placed commonly invoked functions in different compartments to understand the overheads of enabling code compartments. For a fair comparison, we ensured that the number of compartment transitions remained the same in the C and OCaml programs for a given benchmark. In Figure 12, columns marked C + Code comp and OCaml + Code comp denote C and OCaml executables enabled with code compartments, respectively. We can see that the overhead of compartments is low compared to the microbenchmark in the previous section. On average, there is only a 10% increase in execution time when code compartments in

OCaml and C are enabled compared to when compartments are not enabled in the respective languages. Overall, we can see that the overhead FIDES code compartments is low.

The SM stack is used to save and restore the compartment metadata. For inter-compartment C functions, we also save and restore the callee saved register (as we do not trust the callee to preserve the registers). This adds upto 160 bytes, including the metadata, saved for inter-compartment calls from C. OCaml does not have callee saved registers. An inter-compartment call from OCaml saves 64 bytes of metadata on the SM stack. OCaml maintains exception handlers as part of the program stack. Unlike the formal model, FIDES also maintains OCaml exception handlers the same way the OCaml maintains them. So, it does not use a separate exception stack. There is no extra resource consumption for exception stacks.

6.3.2 **Cost of supporting data compartments.** Most of the performance overhead in FIDES is due to the data compartments. This is observed in columns marked C + Fat + Code comp and OCaml + Code comp in Figure 12. In the C version of qsort and crc32, there is a 6× and 19× slowdown (resp.) compared to the C baseline. Interestingly, the OCaml version with compartments enabled is slower on qsort and faster on crc32 compared to the C version with compartments and fat pointers. The choice to switch from an unsafe language like C to a safe language like OCaml is a complex choice that balances many, often conflicting, requirements such as performance, cost of transition, maintainability, richness of the library ecosystem, etc. The qsort result shows that, with FIDES, developers can retain certain parts of the program in C while gaining additional safety and security guarantees, whereas the crc32 result shows that there is, in fact, a clear win in terms of performance when switching to OCaml from C.

6.3.3 **Code size impact.** FIDES introduces two new instructions – checkcap and val – which are introduced in the program. We observed that the introduction of new instructions has a minimal impact on code size. With FIDES, the code size increase is only 4% in C and 2% in OCaml on the benchmark programs in Figure 12.

6.4 Evaluating the EVM application

Our technique scales to real-world applications with significant use of third-party libraries. The EVM application is constructed using 20 existing third-party packages from the MirageOS ecosystem, including mirage-crypto, mirage-runtime, lwt, etc. In total, the EVM application has 68k lines of code (LoC), out of which we wrote 5k lines of new code. 48% of the codebase is in OCaml. For the 68k LoC EVM application, the .cap files and flags in the build scripts specifying the access matrix is 70 LoC. In practice, these annotations only specify entry points into the compartments and default compartment IDs.

Table 4. EVM case study with different compartment (comp) strategies. The baseline for the overhead is the EVM application without FIDES.

# Comp.	Overhead (cycles)	Avg code size / comp (KB)	# Inter-comp trans ($\times 10^4$)
6:NF	1.59×	132	5170
6:F	1.23×	132	5
23:NF	1.60×	47	5320
23:F	1.23×	47	8

We evaluate the overheads of the EVM application with six compartments described in §3. Additionally, we evaluate the same application with another strategy that has 23 compartments, with each OCaml package placed in a different compartment. Further, each compartment strategy is evaluated with (F) and without (NF) fluid compartments.

Table 4 presents the results. Compared to not using FIDES, we observe that FIDES EVM application has 23% overhead in the case of 6:F compartment. Without fluid compartments, the number of inter-compartment transitions increases significantly, which has a corresponding performance drop. When the application is compartmentalized in a fine-grained manner into 23 compartments, we observe that the average compartment code size reduces from 132KB to 47KB. This represents a significantly smaller attack surface and, thus, a more secure application. Interestingly, with fine-grained compartments, the number of transitions does not increase significantly, indicating that logically separate parts of the program have been placed in separate compartments. As a result, the performance remains almost the same. This illustrates that if a security engineer puts in the effort to compartmentalize the application in a fine-grained fashion, then not only is the security improved due to a smaller attack surface, but the performance impact also does not increase significantly compared to coarse-grained compartmentalization. The results also show that fluid compartments play a significant role in keeping the overheads low while providing better security by avoiding the confused deputy attack.

7 Related work

Intra-process compartment techniques have been widely studied over the years. Table 5 compares the recent solutions with respect to their support for safe languages, application in resource-constrained embedded systems, compartment granularity, and sharing data between compartments.

Table 5. Summary of hardware-assisted compartment solutions. ✗/✓: partially supported. †: support multiple compartments within C. **F1**: Maximum number of compartments. **F2**: Support for baremetal embedded systems. **F3**: Support for safe languages. **F4**: Support for fine-grained compartments. **F5**: Support for direct access to shared data.

Technique	F1	F2	F3	F4	F5
Secage[31]	512	✗	✗	✗	✗
Glamdring[30]	2	✗	✗	✗	✗
GOTEE[16]	2	✗	✓	✗	✗
Donky[46]	∞	✗	✗	✗	✗
Enclosures[15]	∞	✗	✓	✗	✗
PKRU-Safe[27]	2	✗	✗/✓	✗	✗
Galeed[44]	2	✗	✗/✓	✗	✗
CHERI-JNI[5]	$2^{\dagger}$	-	✗/✓	✗/✓	✓
ACES[9]	∞	✓	✗	✗	✗
MINION[25]	∞	✓	✗	✗	✗
CHERI-CompartOS[2]	2^{64}	✓	✗	✓	✓
FIDES	256	✓	✓	✓	✓

7.1 Enforcing compartments

Many techniques support compartments on top of existing support for memory protection in commercially available processors. Donky [46] and Enclosures [15] tag pages with compartment IDs to enforce intra-process compartments. Enclosures uses Intel MPK [21] while Donky extends a similar scheme on RISC-V processors. SeCage [31] uses Intel VT-x [53] to enforce isolation between compartments, by setting up separate page tables for each compartment. CETIS [56] utilizes existing Intel CET [21] to support two domains, while Glamdring [30] and GOTEE [16] utilize Intel SGX [22] to partition the application ensuring confidentiality and integrity of sensitive data. All the works discussed above rely on paging and require an OS or a hypervisor. This restricts their applicability in resource-constrained embedded systems which lack paging support. FIDES does not rely on the OS or a hypervisor to enforce compartments, which makes it ideal for memory-constrained baremetal systems. Similar to FIDES solutions like ACES [9] and MINION [25] utilize ARM MPU [3] to enforce compartments and do not rely on paging support.

Capability-based approaches, such as CHERI [55], do not require paging support. They transform every pointer into architectural capabilities to define compartment regions and enforce isolation between them. Compared to FIDES fat pointer scheme, CHERI capabilities are more expressive, as they store extra permission bits (like `rwx`), apart from just base and bounds metadata. These permission bits restrict the operations that can be performed using that capability. Contrary to

CHERI, our goal while developing Fides was to introduce minimal changes to the ISA without affecting the function call and data passing semantics, leveraging the safe language guarantees, thereby making it easier and more straightforward to port a mixed-language application to Fides.

7.2 Support for safe languages

Galeed [44] and PKRU-Safe [27] utilize Intel-MPK to secure the Rust from C/C++ vulnerabilities by splitting the application into two domains, one containing all the Rust code and the other containing all the C code. They do not support compartments within Rust or C/C++ codebase. GOTEE [16] supports compartments in the Go language using Intel SGX. Enclosures [15] provides package-based isolation aimed at sandboxing untrusted third-party packages in Go and Python. All these techniques rely on paging support, restricting their applicability to embedded systems. CHERI-JNI [5] utilizes CHERI capabilities to secure the Java Native Interface [42] but does not support compartments within the Java code. CHERI supports the Rust [47] language but is yet to be ported to garbage-collected languages like OCaml. Fides extensions to the OCaml compiler are lightweight §6.1 and do not require extensive changes to the compiler backend.

7.3 Support for fine-grained compartments

The granularity of the compartment defines the attack surface reduction. Intel-MPK supports 16 compartments and, to support a greater number of compartments, requires software multiplexing, which imposes overheads during a compartment switch [17, 43]. ARM Memory Protection Unit (MPU) based techniques like ACES [9] do not support fine-grained compartments, as the number of regions a compartment can access is limited based on the MPU register count and alignment constraints (currently upto 16). This makes MPU-based techniques unsuitable for protecting multiple separated regions. Fides supports fine-grained compartments, which leads to better attack surface reduction and vulnerability isolation.

7.4 Support for direct access to shared data across compartments

Supporting secure direct data sharing between compartments is critical for performance. Paging-based solutions require OS intervention to tag pages with the same domain ID for sharing between compartments. MPU-based techniques support a limited number of shared regions, restricted by the number of isolated memory regions that can be defined and alignment constraints. GOTEE [16] and Glamdring [30] require deep copying to share data between compartments, changing the semantics of the inter-compartment function calls. Fides utilizes safe language guarantees and hardware-assisted fat pointers to enforce secure direct data sharing between compartments without the need for deep copying or any mediation. Similar to Fides, CHERI also allows direct sharing of data between compartments.

7.5 Support for memory safety in C

Fides builds upon Shakti-MS [11] to enforce memory safety. Fides does not aim to optimise Shakti-MS or propose new memory safety techniques in C. Such optimisations are orthogonal to Fides. Unlike Shakti-MS, we present a formal model of the fat pointer implementation in this paper.

There are many extant works that aim to enforce spatial and temporal memory safety. CCured [39] achieves spatial memory safety by introducing a fat pointer into the language's type system, where the bounds metadata is placed alongside the pointer, similar to Shakti-MS. SoftboundCETS [38] achieves both spatial and temporal memory safety by associating every pointer with bounds and liveness metadata. But unlike CCured, SoftboundCETS stores the metadata in a disjoint look-up table. However, it suffers from significant runtime overhead since before each pointer dereference, the look-up table needs to be accessed to retrieve the metadata for the memory safety checks.

Checked-C [13, 60] is similar to CCured but achieves both spatial and temporal memory safety with the same fat pointer size as CCured has.

Unlike the techniques above, Delta Pointers [28] and Low-Fat [12] ensure spatial memory safety by encoding the bounds metadata within the 64-bit pointer, using compact encoding schemes. ViK[6] enforces temporal memory safety by assigning every heap-allocated object a unique object ID (similar to cookie value in Shakti-MS) and storing these object IDs in the unused pointer bits. The object ID in the pointer is checked to see if it matches the object ID of the object it is accessing. The advantage of the above schemes is that, unlike Shakti-MS, they have a lesser memory footprint, w.r.t pointer representation. The disadvantages of these schemes vary based on their design choices. For example, Low-Fat uses a custom memory allocator which enforces memory alignment constraints for compact bounds encoding. Delta pointers do not prevent buffer underflows, only buffer overflows.

Contrary to changing pointer representation, techniques like DangNull [29] and FreeSentry [58] enforce temporal memory safety by assigning all pointers to a freed memory object a NULL value, thereby eliminating dangling pointers. Techniques like MarkUs [1] and Dieharder [41] utilize custom memory allocators to enforce temporal memory safety. They take advantage of the huge virtual address space to ensure that a freed memory region is not immediately reallocated. The major disadvantage of these approaches is their huge memory overheads, which makes them impractical for resource-constrained environments.

7.6 Formal semantics of compartments

In §4, we formalise the semantics of compartments in the presence of tail-calls, higher-order functions and exceptions, and memory safety in a multi-language setting with the help of fat pointers. MSWasm [35] extends the WebAssembly [18] with custom memory safety instructions. MSWasm introduces a colour-based memory-safety monitor and shows that MSWasm is memory-safe. In addition, they formalise a compilation scheme from a minimal idealised subset of C to MSWasm and prove that the compiler enforces memory safety. In this work, we do not formalise memory safety but observe that MSWasm's colour-based memory safety can be directly applied to FIDES.

SECOMP [24, 52] introduces secure compartmentalizing compilation and extends the CompCert-verified C compiler with support for compartments that target the CHERI capability machine. Unlike FIDES, SECOMP does not support tail calls and exceptions, and provides no special support for higher-order functions. We do not formalise the compilation of C to FIDES. We leave this to future work. Recent work [59] has also mechanised the formal semantics of CHERI dialect of C, clarifying the behaviour of capabilities and undefined behaviours. In this work, we do not focus on the source language but formalise the semantics of the hardware and the ABI.

8 Conclusion

In this work, we have presented FIDES, which offers intra-process compartments for applications that mix safe and unsafe languages and deploy untrusted third-party libraries. FIDES specifically handles language features commonly found in high-level safe languages, such as higher-order functions, tail calls and exceptions. Given the increasing awareness of the threats of unsafe languages [20] and the impossibility of a wholesale move of large legacy codebases to a safe language, we believe that FIDES will provide an important stepping stone that will ease such a move.

References

[1] Sam Ainsworth and Timothy M. Jones. 2020. MarkUs: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)*. 578–591. https://doi.org/10.1109/SP40000.2020.00058

[2] Hesham Almatary. 2022. *CHERI compartmentalisation for embedded systems.* Technical Report.
[3] Ying Bai. 2016. *ARM® Memory Protection Unit (MPU).* 951–974. https://doi.org/10.1002/9781119058397.ch12
[4] Olivier Benot. 2011. *Fault Attack.* Springer US, Boston, MA, 452–453. https://doi.org/10.1007/978-1-4419-5906-5_505
[5] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. 2017. CHERI JNI: Sinking the Java Security Model into the C. *SIGARCH Comput. Archit. News* 45, 1 (apr 2017), 569–583. https://doi.org/10.1145/3093337.3037725
[6] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2022. ViK: practical mitigation of temporal memory safety violations through object ID inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 271–284. https://doi.org/10.1145/3503222.3507780
[7] Catalin Cimpanu. 2018. *Twelve malicious Python libraries found and removed from PyPI.* https://www.zdnet.com/article/twelve-malicious-python-libraries-found-and-removed-from-pypi/
[8] Catalin Cimpanu. 2019. *Two malicious Python libraries found and removed from PyPI.* https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/
[9] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. ACES: Automatic Compartments for Embedded Systems. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 65–82. https://www.usenix.org/conference/usenixsecurity18/presentation/clements
[10] Mitre Corporation. 2023. *Common Weakness Enumeration, A Community-Developed List of Software and Hardware Weakness Types.* https://cwe.mitre.org/
[11] Sourav Das, R. Harikrishnan Unnithan, Arjun Menon, Chester Rebeiro, and Kamakoti Veezhinathan. 2019. SHAKTI-MS: A RISC-V Processor for Memory Safety in C. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Phoenix, AZ, USA) *(LCTES 2019)*. Association for Computing Machinery, New York, NY, USA, 19–32. https://doi.org/10.1145/3316482.3326356
[12] Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. *Proceedings of the 25th International Conference on Compiler Construction* (2016). https://api.semanticscholar.org/CorpusID:15649474
[13] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*. 53–60. https://doi.org/10.1109/SecDev.2018.00015
[14] Neel Gala, Arjun Menon, Rahul Bodduna, G. S. Madhusudan, and V. Kamakoti. 2016. SHAKTI Processors: An Open-Source Hardware Initiative. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*. 7–8. https://doi.org/10.1109/VLSID.2016.130
[15] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/3445814.3446728
[16] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-Based Construction of Trusted Execution Environments. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 571–585.
[17] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 609–624. https://www.usenix.org/conference/atc22/presentation/gu-jinyu
[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363
[19] Norm Hardy. 1988. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.* 22, 4 (oct 1988), 36–38. https://doi.org/10.1145/54289.871709
[20] White House. 2024. *Back to the building blocks: A path toward secure and measurable software.* https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
[21] Intel. [n. d.]. *Intel® 64 and IA-32 Architectures Software Developer Manuals.* https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
[22] Intel. 2014. *Intel Software Guard Extensions Programming Reference.* https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf
[23] RISC-V International. 2021. *RISC-V International.* https://riscv.org/
[24] Roberto Blanco Aïna Linn Georges Cătălin Hritcu Andrew Tolmach Jérémy Thibault, Arthur Azevedo de Amorim. 2023. *SECOMP2CHERI: Securely Compiling Compartments from CompCert C to a Capability Machine.* https://catalin-

hritcu.github.io/publications/SECOMP2CHERI-PriSC23.pdf

[25] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, X. Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *Network and Distributed System Security Symposium*.

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 361–372.

[27] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 132–148. https://doi.org/10.1145/3492321.3519582

[28] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta pointers: buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/3190508.3190553

[29] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Network and Distributed System Security Symposium*. https://api.semanticscholar.org/CorpusID:6119920

[30] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 285–298.

[31] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[32] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A Survey of Microarchitectural Side-Channel Vulnerabilities, Attacks, and Defenses in Cryptography. *ACM Comput. Surv.* 54, 6, Article 122 (jul 2021), 37 pages. https://doi.org/10.1145/3456629

[33] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[34] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-259/

[35] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. 2023. MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code. *Proc. ACM Program. Lang.* 7, POPL, Article 15 (jan 2023), 30 pages. https://doi.org/10.1145/3571208

[36] Mozilla. 2024. *How much Rust in Firefox?* https://4e6.github.io/firefox-lang-stats/?2022-03

[37] MozillaWiki. 2020. *Oxidation*. https://wiki.mozilla.org/Oxidation

[38] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[39] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (may 2005), 477–526. https://doi.org/10.1145/1065887.1065892

[40] NIST. 2020. *CVE-2020-35511*. https://nvd.nist.gov/vuln/detail/CVE-2020-35511

[41] Gene Novark and Emery D. Berger. 2010. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. Association for Computing Machinery, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

[42] Oracle. 2023. *Oracle Java Native Interface*. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html#java_native_interface_overview

[43] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: software abstraction for intel memory protection keys (intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 241–254.

[44] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 824–836. https://doi.org/10.1145/3485832.3485903

[45] J.H. Saltzer and M.D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[46] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient in-Process Isolation for RISC-V and X86. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 95, 18 pages.

[47] Nicholas Wei Sheng Sim. 2020. *Strengthening memory safety in Rust: exploring CHERI capabilities for a safe language.* Master's thesis. https://nw0.github.io/cheri-rust.pdf

[48] Sergio De Simone. 2022. *Linux 6.1 Officially Adds Support for Rust in the Kernel.* https://www.infoq.com/news/2022/12/linux-6-1-rust/

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

[50] Benchmarksgame Team. 2023. *The Computer Language 23.03 Benchmarks Game: Which programming language is fastest?* https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[51] MSRC Team. 2019. *A proactive approach to more secure code.* https://msrc.microsoft.com/blog/2019/07/16/a-proactive-approach-to-more-secure-code/

[52] Jérémy Thibault, Roberto Blanco, Dongjae Lee, Sven Argo, Arthur Azevedo de Amorim, Aïna Linn Georges, Catalin Hritcu, and Andrew Tolmach. 2024. SECOMP: Formally Secure Compilation of Compartmentalized C Programs. arXiv:2401.16277 [cs.PL] https://arxiv.org/abs/2401.16277

[53] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56. https://doi.org/10.1109/MC.2005.163

[54] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting Software with Code-Centric Memory Domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 469–480.

[55] 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. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. 20–37. https://doi.org/10.1109/SP.2015.9

[56] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2022. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2989–3002. https://doi.org/10.1145/3548606.3559344

[57] Xilinx. 2022. *AMD Artix 7 FPGA AC701 Evaluation Kit.* https://www.xilinx.com/products/boards-and-kits/ek-a7-ac701-g.html#overview

[58] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Network and Distributed System Security Symposium.* https://api.semanticscholar.org/CorpusID:2297018

[59] Vadim Zaliva, Kayvan Memarian, Ricardo Almeida, Jessica Clarke, Brooks Davis, Alexander Richardson, David Chisnall, Brian Campbell, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2024. Formal Mechanised Semantics of CHERI C: Capabilities, Undefined Behaviour, and Provenance. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 181–196. https://doi.org/10.1145/3617232.3624859

[60] Jie Zhou, John Criswell, and Michael Hicks. 2023. Fat Pointers for Temporal Memory Safety of C. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 86 (apr 2023), 32 pages. https://doi.org/10.1145/3586038