FIDES: End-to-end Compartments for Mixed-language Systems

Sai Venkata Krishnan
Indian Institute of Technology Madras
Chennai, India
saiganesha5.svkv@gmail.com

Arjun Menon
InCore Semiconductors
Chennai, India
arjun@incoresemi.com

Chester Rebeiro
Indian Institute of Technology Madras
Chennai, India
chester@cse.iitm.ac.in

KC Sivaramakrishnan
Indian Institute of Technology Madras and Tarides
Chennai, India
kcsrk@cse.iitm.ac.in

Abstract

Memory-unsafe code and monolithic designs remain major sources of vulnerabilities in embedded systems. Although developers are increasingly adopting memory-safe languages such as Rust and OCaml, mixed-language applications still rely on C libraries, which weakens security boundaries. Existing compartment schemes are built for C and cannot accommodate essential high-level language features such as higher-order functions, closures, exceptions and tail-call optimisation. This makes them unsuitable for modern safe-language ecosystems.

Fides addresses these limitations with a lightweight hardware-assisted compartment scheme designed for end-to-end compartmentalisation of mixed OCaml–C applications. Fides enforces fine-grained, function-level isolation, preserves advanced control-flow behaviour, and leverages the guarantees of a memory-safe language to enable safe data sharing across compartments. To accommodate closures and higher-order functions, Fides introduces a novel *fluid* compartment scheme that permit secure, flexible sharing of closure environments without enlarging attack surfaces. To support unsafe C components, Fides uses hardware-assisted fat pointers that enforce spatial and temporal safety, and leaves OCaml pointers untouched so that safe-language code incurs zero instrumentation and retains full memory safety. We implement Fides on a modified RISC-V processor and evaluate it on MirageOS unikernels. Fides executes OCaml code with no additional overhead and adds only a modest cost to C code. Case studies on a MirageOS HTTPS server and an electronic voting machine show that Fides provides strong isolation for real embedded workloads with modest engineering effort.

ACM Reference Format:
Sai Venkata Krishnan, Arjun Menon, Chester Rebeiro, and KC Sivaramakrishnan. 2026. FIDES: End-to-end Compartments for Mixed-language Systems. In . ACM, New York, NY, USA, 15 pages.

Conference'17, Washington, DC, USA

1 Introduction

Embedded and IoT devices increasingly run security-critical software yet are difficult to patch, making them attractive targets [30]. A major driver of vulnerabilities is memory-unsafe C/C++ code, which remains pervasive in embedded systems [28, 68]. As a result, many companies are migrating toward memory-safe languages. Safe[1] languages, such as OCaml, Rust, Java, JavaScript and Python, rely on language semantics and compiler/runtime mechanisms to prevent memory corruption. Major projects, such as Linux [65] and Mozilla Firefox [52], are incorporating safe-language code, and more than 80% of embedded companies have started adopting safe languages [8].

Yet memory safety is not the full story. First, real systems are mixed-language: safe-language components routinely depend on legacy C libraries (e.g., cryptography, device drivers, parsers), and a single unsafe component can still compromise the whole address space. Second, even in safe languages, vulnerabilities and malicious third-party packages can leak secrets or trigger unintended behaviour without relying on memory corruption [10, 11, 14]. What makes such failures catastrophic is the monolithic design: code with different trust assumptions shares a single address space, so a bug in an untrusted component can pivot to compromise sensitive state and enable broader attacks [54, 56].

Software compartmentalisation [13, 23, 24, 35, 43, 44, 63, 72] addresses this risk by partitioning an application into isolated components that execute with least privilege [62]. Compartments permit sensitive code to be isolated from the rest of the application, limiting the impact of a vulnerability in an unsafe component or a malicious third-party library [10, 11]. In a typical scheme, each component has a *code compartment* and a *data compartment*; access policies constrain inter-compartment control flow and, indirectly or explicitly, the data that each compartment can reach.

We observe that current software compartmentalisation schemes [3, 13] are designed specifically for C and C++ software. With the growing traction towards safe languages, we increasingly encounter applications that mix safe and unsafe code. Existing schemes cannot be directly applied to such mixed-language applications. There have been attempts to compartmentalise mixed-language applications, such as CHERI-JNI [9, 60]. However, these primarily aim to isolate safe-language runtimes from unsafe native code and prevent cross-language attacks [50], rather than providing end-to-end

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

compartmentalisation across the whole application. Designing a fine-grained compartment scheme for mixed-language applications remains challenging due to the following reasons:

- **C1** Safe languages and unsafe languages provide different security guarantees. The compartment scheme should take these differences into account and ensure that the scheme's security guarantees are uniform across languages.
- **C2** Higher-order functions and function closures are widely used in safe languages like Java, OCaml and Python and modern C++ (via lambdas). 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 compartmentalisation 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 compartmentalisation 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 mixed-language applications. In this work, we use OCaml as the safe language and C as the unsafe language.

Code compartments. To realise code compartments, Fides extends both the C and OCaml toolchains to consume a *policy file* that assigns functions to compartments, defines inter-compartment access policies, and specifies distinguished functions as valid entry points. Control flow within a compartment is unrestricted. Cross-compartment control flow is allowed only if the access policy permits it and only through valid entry points. Fides enforces these checks efficiently with hardware assistance.

Data compartments. Fine-grained data sharing across compartments is hard in conventional C-centric schemes: sharing often requires either broadening access (increasing attack surface) or copying (changing semantics and adding overhead). Fides leverages safe-language guarantees to simplify this design space. A type-safe OCaml program is memory-safe, and the data that an OCaml function can access is precisely the set of objects transitively reachable from its locals, arguments, and globals. Fides preserves this property in mixed OCaml–C applications and, to address **C1**, hardens C with spatial and temporal memory safety using hardware-assisted fat pointers [17, 53, 55, 71, 74], while leaving OCaml pointers untouched.

To address **C2**, Fides introduces the notion of *fluid compartments* that facilitates flexible compartment strategies to securely share closures between compartments without compromising security. We design our compartment scheme to preserve tail-calls and support exceptions (challenge **C3**).

Fides does not rely on an OS or MMU, making it suitable for constrained embedded systems that do not support either. We run OCaml+C based MirageOS [47] unikernels *baremetal* on a modified RISC-V [34] processor. Fides allows compartment access policies to be defined separately from the source code, enabling a security engineer to compartmentalise applications that include untrusted third-party code.

Compared to CHERI [71] based schemes, Fides does not require tagged memory, which makes Fides better suited for power-constrained devices. Extending CHERI compartmentalisation to safe languages [64] requires substantial compiler and toolchain support. Fides instead introduces a lightweight design with only two custom instructions: (1) checkcap for code compartments and (2) val for 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 an implementation of Fides on a modified Shakti RISC-V processor [21] that executes baremetal MirageOS [47] unikernels (§5).
- We demonstrate the effectiveness of Fides with a security-critical electronic voting machine (EVM) application, and a HTTPS web server. Our evaluation of Fides on the Xilinx Artix-7 AC701 FPGA [73] shows that Fides offers an attractive security-performance tradeoff (§6).

2 Background

In this section, we provide brief background on (i) memory-safe languages and (ii) functional-language features that interact with compartmentalisation.

2.1 Memory-safe languages

Memory-safe languages such as OCaml [42] and Rust [49] restrict low-level memory manipulation in their safe subsets. They combine static and runtime techniques—for example, ownership/borrowing in Rust [39] and garbage collection in OCaml [46] (and Go [25])—to provide spatial memory safety and to prevent common temporal errors in safe code. Compared to C and C++, these languages eliminate large classes of memory-safety vulnerabilities, substantially reducing the attack surface for memory-corruption exploits. Many memory-safe languages are also statically typed and restrict unchecked casts in safe code, which helps preserve type-based invariants and reduces the risk of type confusion [12, 40].

Memory-safe languages are increasingly adopted in real systems. For example, Firefox incorporates Rust [51], and Docker Desktop has long used OCaml unikernels in production [48]. Clean-slate unikernels [16] such as MirageOS [47] (OCaml) and RustyHermit [41] (Rust) are implemented primarily in memory-safe languages.

2.2 Functional programming paradigms

Functional programming features, such as higher-order functions and tail-call optimisation, are widely used in OCaml and are increasingly common in mainstream languages (e.g., C++, JavaScript, Python, and Java). Similar "behaviour-as-data" idioms also appear in object-oriented code via callback objects and patterns such as Strategy and Visitor [22]. These features are central to Fides's design because they affect both control flow and the placement and sharing of closures across compartments.

Higher-order functions. In functional programming languages, functions are treated as first-class values and can be passed around in the same way as primitive values can be. Higher-order functions (HoFs) take one or more functions as parameters and may also return a function. These functions may also capture variables in scope via *closures*. In object-oriented languages, passing a closure is closely analogous to passing an object that implements a single-method interface (often used to encode callbacks and patterns such as Strategy or Visitor [22]), where the captured variables correspond to the object's fields. HoFs are directly supported in functional programming languages like OCaml and are increasingly adopted in mainstream languages, such as C++ [1] and Java [18]. The code listing below shows an example of a HoF written in OCaml that allocates a closure.

```
1  let res_tab = Array.make num_candidates 0
2  let count_votes votes_arr =
3    let inc_count cand_id =
4      res_tab.(cand_id) <- res_tab.(cand_id) + 1
5    in
6    Array.iter inc_count votes_arr
```

The function `inc_count` captures `res_tab` in its closure environment, and is passed to `Array.iter`, which is a HoF that iterates over all elements of `votes_arr` and calls `inc_count` on each element. This simplified pattern is the same as the one we later use to motivate fluid compartments (§4.4).

Tail-call optimisation. Compilers often employ tail-call optimisation (TCO) to avoid stack growth by reusing the caller's stack frame when the last operation in a function is a call. Tail calls commonly appear as tail recursion and as tail calls across distinct functions (including mutually recursive functions). TCO is widely used in functional programs, where recursion is often used to encode iteration, and it also appears in asynchronous programming and continuation-passing style code to invoke callbacks efficiently [4, 70].

```
1  let foo x = x + 1
2  let bar x = foo (x + 7)
```

In the code listing above, the last operation in the function `bar` is a call to the `foo` function. This call is eligible for TCO and can be compiled as a tail call. In practice, this matters because the same tail-call pattern occurs in tail-recursive loops and continuation-passing style code; without TCO, repeated tail calls would grow the stack.

Exceptions. OCaml exceptions are known to be fast and are used for control-flow in many libraries. For example, returning early from a HoF is often implemented by raising an exception that is caught outside the HoF. Hence, supporting exceptions is crucial for Fides to be practical. Exceptions provide non-local control flow: raising an exception may unwind multiple stack frames until a suitable handler is found. This matters for compartmentalisation because it creates implicit cross-function transfers of control that must remain well-defined and policy-compliant even when unwinding crosses compartment boundaries.

3 Threat model

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

Table 1: Common Weakness Enumeration (CWE) [15] that Fides mitigates or significantly weakens the damage

CWE	C+OCaml	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	●	●	●
Privilege-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

3.1 Assumptions and limitations

Fides permits applications to be built with a mix of safe (OCaml) and unsafe (C) code. The application may contain malicious, untrusted third-party libraries, and the attacker may supply arbitrary inputs to the application and attempt to trigger vulnerable behaviors. The attacker has full knowledge of the internals of Fides, the application's source code, and the compartment access policy. We assume the Fides hardware extensions and the Security Monitor (SM) are trusted and correctly enforce the compartment policy. 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, a statically linked binary, cannot be tampered with. Hardware- [36], fault- [7] and side-channel attacks [45] are beyond the scope of the model.

3.2 Attacks

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

3.2.1 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 [66].

```
1  let admin_flag = ref false
```

```
2  ...
3  if !admin_flag then
4    (* do privileged operation *)
```

In the code above, the attacker may exploit a memory error-based CWE in C (Table 1) to update `admin_flag` in OCaml to `true`, thereby performing privileged operations. Appendix A (§8) in the supplementary material presents the source code for an attack that uses an out-of-bounds write in C to update the `admin_flag` in OCaml. Making C memory-safe helps thwart memory error-based CWEs. Fides thwarts this attack with the help of hardware-assisted fat pointers for C, which provides spatial and temporal memory safety for C.

3.2.2 Isolation. Our aim is to build secure MirageOS unikernels [47] 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 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.

```
1  value *callback_sum = caml_named_value("sum");
2  value *callback_leak =
       caml_named_value("leak");
3  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 (§B) in the supplementary material shows the entire working example. Fides provides the compartment mechanism for specifying and restricting unintended control flow in the program, thereby helping to prevent control flow subversion. We shall discuss the details of the mechanisms in the next section.

4 Fides Design

In this section, we describe the design of Fides and demonstrate its effectiveness by securing an electronic voting machine (EVM) application implemented as a MirageOS unikernel. We also explain how Fides supports compartmentalisation in a mixed-language codebase and in the presence of higher-order functions, tail-call optimisation and exceptions.

4.1 Compartments with Fides

Compartmentalising an application involves partitioning code and data into isolated components with explicit access policies. We first explain how Fides enforces code and data compartments for the safe-language (OCaml) part of an application, and then how it extends these guarantees to the unsafe C portion.

4.1.1 **Code compartments.** Each code compartment is a set of functions. Any control-flow transfer within a compartment is permitted. When control flows between functions in different compartments, Fides enforces the compartment policy. We define the access-control policy as a set of directed compartment pairs (A, B), meaning that functions in compartment A are permitted to transfer control to functions in compartment B. Fides supports up to 256 code compartments and permits function-level compartment assignments.

We first look at the steps to create code compartments. Importantly, these steps are driven by a separate policy and are enforced by the toolchain at build time (compile/link time), without requiring any changes to the application source code.

Step 1: Map every function to a compartment. For every source code file in the application, Fides expects a separate `.cap` policy file that contains the function-to-compartment mapping. This policy is consumed by the toolchain and enforced at link time, so the application code can be developed independently of the compartmentalisation strategy (i.e., no source-code annotations or refactoring are required). For example, in the listing below, `get_key` is mapped to compartment **CP2**.

```
<function>:<compartment id>:<external>
count_tally     : CP2 : ENTRY_POINT
get_key         : CP2 : NO_ENTRY_POINT
password_check  : CP2 : NO_ENTRY_POINT
sanitize_input  : CP2 : ENTRY_POINT
```

Fides uses a custom linker script to group all functions belonging to the same compartment together in memory at link time. As a result, the code belonging to a particular compartment resides within a non-overlapping, contiguous address range.

Step 2: Assign compartment entry points. Compartment entry points are the only valid targets of cross-compartment function calls. In the policy file (Listing 4.1.1), these functions are marked as `ENTRY_POINT`. To identify entry points at runtime, the Fides toolchain inserts a custom instruction checkcap at compile time as the first instruction of each entry-point function. The hardware rejects any cross-compartment call/jump whose target is not a valid entry point.

Step 3: Assign compartment access policy to each compartment. The compartment access policy is encoded as an $N \times N$ Boolean matrix, where N is the number of compartments. The access policy captures the allowed cross-compartment calls; the return path back to the caller is implicit. For example, consider the following configuration: $AccessPolicy[2, 3] = 1$ and $AccessPolicy[3, 2] = 0$. This means that a function in compartment **CP2** is permitted to call a function in **CP3**, and control may return from **CP3** to **CP2**. However, a call originating from **CP3** to **CP2** is not allowed. The access policy is a runtime structure stored in a part of memory inaccessible from the user program. All the above steps are driven by a security engineer-provided policy and enforced by the toolchain at build time.

We now see how Fides enforces code compartments at runtime. Fides extends the processor pipeline and tracks the currently executing compartment *context*, which includes the compartment

identifier and the corresponding code region's start and end addresses. While executing each instruction, the hardware uses the current value of the program counter (*pc*) to check that the instruction lies within the current compartment boundary. Whenever a control-flow transfer crosses a compartment boundary, the hardware first checks if the target of the call/jump instruction is a valid entry point, i.e., an checkcap instruction. If not, the transition fails and execution is terminated.

If it is a valid entry point, execution traps to a custom trap handler – Security Monitor (SM). The SM is a trusted module that is statically linked to every application compiled with the Fides compiler toolchain. The SM **(a)** identifies the caller and callee compartments, and **(b)** checks the compartment access policy. If access is allowed, the SM saves the current compartment context onto a private, shadow SM stack (inaccessible from user code) and updates the compartment *context* to that of the target compartment.

To identify cross-compartment returns, Fides saves the return address on the SM stack and overwrites the return address on the program stack with a magic value μ outside of the executable code region. On return, since μ is not a valid program address, execution traps to the SM. The SM restores the previous compartment context from its private stack and resumes execution at the saved return address. This mechanism also prevents ROP[61] attacks in which the callee compartment corrupts the return address on the program stack. To prevent SM memory from being tampered with, the SM code is placed in an isolated compartment with no explicit cross-compartment transitions allowed to it.

4.1.2 **Data compartments.** Generally, in contemporary compartment schemes, such as ACES [13] and Donky [63], explicit data compartments are achieved by annotating data regions that a compartment can access. Fides achieves data compartments implicitly by leveraging the fine-grained memory safety guarantees that the safe language provides.

In a memory-safe language [46], an attacker cannot arbitrarily corrupt memory to forge new references to protected data. From a given *pc*, the data regions that an attacker can access are the transitive closure of all reachable memory regions starting from the currently active stack frame, global data region, and heap. Combining this with the code compartments that we explained before, we implicitly enforce data compartments: the set of data regions accessible from a given *pc* is further constrained by restricting control flow to respect the compartment access policy specified by the security engineer.

In Fides, all compartments share the same stack and heap memory, so data sharing between compartments follows the language semantics and does not require deep copying or explicit shared regions [13, 23, 63].

4.2 Extending Fides to C code

To enforce the compartment view to C code, Fides extends the same code compartment techniques to the C compiler. To enforce data compartments in the C code by ensuring memory safety, Fides adopts a compiler- and hardware-assisted memory safety scheme.

Fat-pointer scheme. For every memory object allocated on stack or heap, we track the object's `Base` address and the `Size` of allocation in a disjoint metadata table (Figure 2). Fides targets a 32-bit address space on a 64-bit ISA, leaving unused upper pointer bits to store fat-pointer metadata. In the unused top bits of each pointer, we encode (i) the `Index` of that object's entry in the metadata table and (ii) a random number called `Cookie`. The `Index` width is currently fixed at 16 bits, implying up to 65K entries can be accommodated in the table. The `Cookie` is used to enforce temporal safety. The remaining 8 unused bits in the pointer are reserved for future use.

Compiler instrumentation. Our custom compiler pass intercepts every stack/heap allocation, creating an entry in the metadata table and converting the pointer into the corresponding fat pointer by populating the `Index` and `Cookie` fields. Similarly, when the metadata table entry is deallocated, it is zeroed out.

Memory safety checks. To ensure memory safety at runtime, we introduce a custom instruction: `val <fat-pointer>`. This instruction is inserted by the compiler just before every fat-pointer dereference. At runtime, the `val` instruction fetches the metadata (`base, size, cookie`) stored at the particular `index` in the disjoint metadata table. To ensure:

- Spatial memory safety, the pointer address is checked to be within the range [`base`, `base+size`).
- Temporal memory safety, it checks that the `cookie` field within the fat pointer matches the cookie value retrieved from the metadata table.

If any of the above checks fail, a trap occurs and execution is terminated (fail-stop). Following the `val` instruction is the actual `load/store` instruction. We extend the hardware to mask the top metadata bits in the fat-pointer before issuing the memory fetch. This is similar to ARM Top Byte Ignore[5], Intel Linear Address Masking[31].

FFI Interaction. Fides depends on type-safety to guarantee memory safety for the safe language. Hence, fat pointers are not used on the safe language side, avoiding the overhead. However, this makes FFI challenging. OCaml has a rich FFI that permits C code to read and write OCaml objects. Without guardrails, a malicious C library can violate the OCaml guarantees. To enforce the fat-pointer scheme for OCaml objects passed to C, we use the object-size metadata stored in the OCaml object header to craft fat pointers (bounds metadata and a random cookie) at the OCaml–C boundary, and remove the corresponding entries from the metadata table when returning to OCaml.

Why disjoint metadata table? Our design decision of adopting a disjoint metadata scheme compared to an in-place metadata scheme like CHERI [71] is multi-fold:

(1) Pointer width stays the same across C and OCaml, which makes FFI interaction straightforward.
(2) This metadata is privileged and inaccessible from user code.
(3) To ensure the integrity of in-place metadata stored in a fat pointer, especially in composite data structures like unions/structures, CHERI relies on tagged memory. Fides does not rely on tagged memory.

Security analysis. Despite storing the bounds information in a disjoint metadata table, there is still a possibility of the `Index`

Table 2: Compartments in EVM

ID	Name	Description
CP1	main-menu	Handles main menu and drives EVM application.
CP2	admin	All code which requires election official privilege.
CP3	crypto	C cryptographic libraries with OCaml wrappers.
CP4	votes-handler	Performs voter validation. Reads plaintext vote from user, encrypts using crypto library, and stores it in 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 module. Contains third-party libraries including C code. **Untrusted.**
CP6	ocaml-stdlib	OCaml standard library and the OCaml runtime.

and `Cookie`, stored in the pointer, being corrupted by an attacker. To mount a successful attack, the attacker has to corrupt both the `Index` and `Cookie`, along with the `Address` field, so that the `val` instruction checks pass. Since the `Cookie` field is randomly assigned at allocation, the attacker can only guess the value so that the corrupted `Cookie` field matches the cookie stored in the metadata table entry of the object the attacker wishes to access. With an 8-bit cookie, the probability of a successful guess in a single attempt is 2^{-8}; after q independent online attempts, the success probability grows to $1-(1-2^{-8})^q \approx q \cdot 2^{-8}$. In our setting, incorrect guesses are fail-stop: a failed `val` check traps and terminates execution, limiting online guessing attempts.

Limitations.

- The Fides fat-pointer scheme is designed for a 64-bit ISA, assuming a 32-bit address space. The 4 GB memory limit is not a limitation in embedded systems, which typically have limited memory resources.
- Currently, the metadata table entry format 2 is designed in such a way that we require only one memory read to fetch the complete metadata entry while executing the `val` instruction. This, however, comes at the cost of restricting the `Cookie` to 8 bits and the `Size` field to 24 bits. Due to this, we support object sizes of up to 16 MB for allocation. In all the experiments we performed, we didn't cross an object allocation size of 16 MB. We can increase the `Size` field, but at the cost of crossing the 64-bit metadata entry size, which would result in an additional memory read.
- We restrict the `Index` field stored in the top bits of the pointer to 16 bits. In many embedded system programs, the number of memory objects that are live at any given time is small, and 16 bits can represent 65K live objects. In future, the unused 8 bits in the pointer can be repurposed as `Index` bits based on the application usage.

4.3 Securing EVM with Fides

EVM design. 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

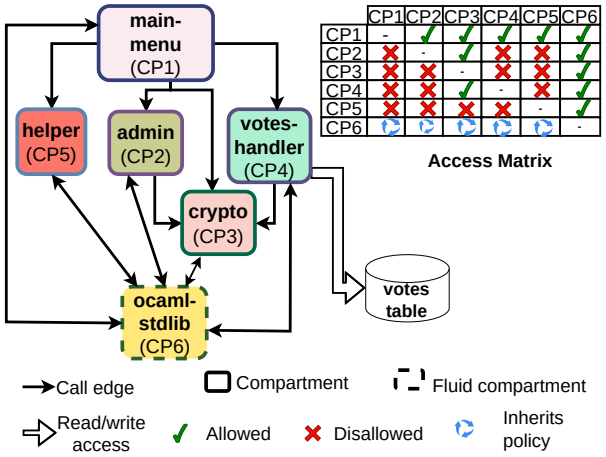

Figure 1: EVM application: The edges between compartments depict the permitted control flow. Only CP4 has access to the votes table. The access matrix lists all allowed compartment transitions.

ID. The vote is then read, 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 credentials to decrypt and count the votes. The encryption is performed using an AES implementation in C.

Security objectives of EVM application. Our goal is to prevent invalid votes, double voting, and leaking votes before the counting day. We show how Fides secures the EVM application and achieves its security goals.

We use Fides to secure the EVM application and achieve its security goals. Figure 1 shows the high-level design of the compartmentalised EVM application. We compartmentalise the application based on whether it requires an election official's credentials 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 it does not comply with the defined access policy.

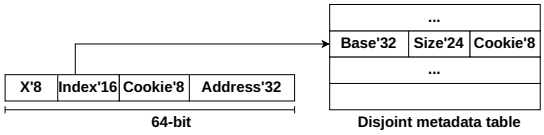

Figure 2: Fat-pointers that define the range and validity of a memory pointer used in C programs.

4.4 Fluid compartments

Defining compartment policies for higher-order function closures is a challenging task. For example, consider the following code:

```
let res_tab = Array.make num_candidates 0
let count_votes votes_arr =
  let inc_count cand_id =
    res_tab.(cand_id) <- res_tab.(cand_id) + 1
  in
  Array.iter inc_count votes_arr
```

The code above belongs to the vote counting module in compartment **CP4**. The array `res_tab` is a table that contains the election results, mapping candidate IDs (from 0 to $N-1$ for N candidates) to the number of votes cast for each candidate. The counts are initialised to 0. `votes_arr` is the array of votes in plaintext. Observe that the HoF `inc_count` 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` function be placed in so that the security guarantees are preserved and 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_count`, 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**.

Standard library functions, such as `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_count`'s compartment **CP4**. Additionally, this scheme also raises security concerns since compartment **CP6** is granted 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* [27] as shown on the left of Figure 3. The attacker in the untrusted **CP5** compartment can call `Array.iter` with `inc_count` 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 HoFs.

To securely and efficiently compartmentalise HoFs, 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 3 shows **CP6** marked as a fluid compartment. When the attacker invokes `Array.iter` with `inc_count` 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_count` fails. This prevents confused deputy attacks.

Fides treats all control transfers to fluid compartments as intra-compartment calls. Our results in §6 show that the cost of switching to a fluid compartment is closer to an intra-compartment call and

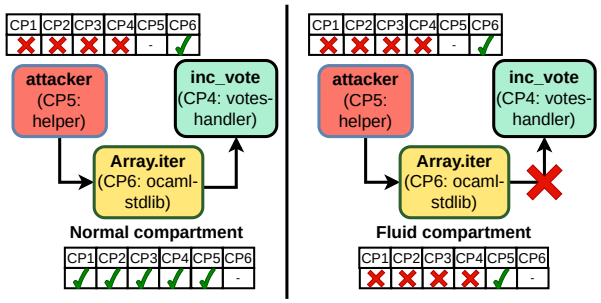

Figure 3: CP6 implemented as a normal compartment (Left) vs. a fluid compartment (right).

is much cheaper than an inter-compartment call. With fluid compartments, Fides enables security engineers to develop secure and cost-effective compartmentalisation schemes.

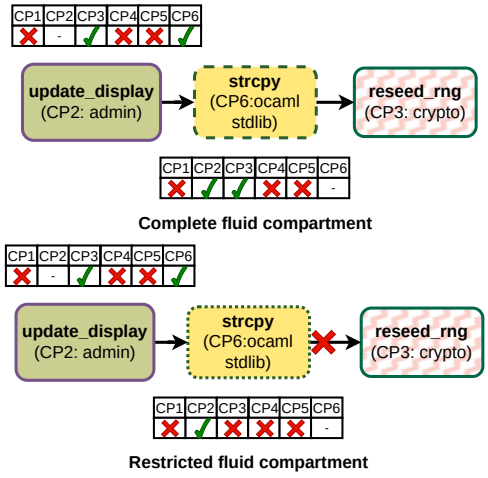

Figure 4: Fluid compartment types

Restricted fluid compartment. For first-order functions from the standard library, such as `strcpy`, we would like to share them across compartments without incurring the overhead of cross-compartment calls. However, using the fluid compartment mechanism as-is may lead to situations where they may be over-privileged. Since the fluid compartment functions inherit the caller compartment's privileges, they can access all the compartments that the caller compartment can access. However, this privilege is unnecessary; the first-order functions, which do not take function pointers as arguments, only need to access their own compartment's data and code.

Worse, this can lead to exploits. Consider the case in the EVM application shown on the left of Figure 4. 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

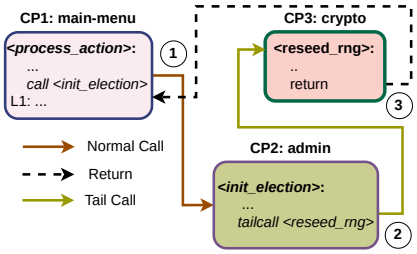

Figure 5: Fides support for tail-calls across compartments. After the tail-call from CP2 to CP3, Fides ensures that control returns to CP1 skipping CP2

misuse this over-privilege to redirect control to the **crypto** compartment and manipulate sensitive cryptographic states. 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 caller compartment. The choice to allow calls to the caller compartment is to permit higher-order callbacks, which are pervasive in OCaml. Observe that when **CP6** is marked as a restricted fluid compartment (RHS of Figure 4), `strcpy` can no longer access the crypto compartment.

4.5 Supporting tail-calls

Existing code compartment schemes support typical call and return sequences. Higher-level languages such as OCaml support non-local control-flow abstractions such as exceptions and include compiler optimisations that break simple call-return sequences, notably tail calls. Since tail-call optimisation (TCO) is pervasive, any practical compartmentalisation system must preserve tail-call semantics.

However, TCO conflicts with the SM stack behaviour. Consider the example given in Figure 5. The function `process_action` calls `init_election`, which in turn calls `reseed_rng`, all mapped to different compartments. Since the last instruction in `init_election` is a callto `reseed_rng`, the compiler applies TCO. The function `reseed_rng` returns to `process_action`, skipping `init_election` in the return path. 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 and maintained in sync with the program call stack. Otherwise, it would result in improper configuration of the compartment *context*, crashing the program. In the example, when `reseed_rng` returns, the compartment *context* of **CP1** should be configured and not **CP2**.

A trivial solution is to disable TCO completely. In OCaml, iteration is often written with tail recursion, with the guarantee that the compiler will transform the tail recursion into a loop. Disabling TCO breaks this fundamental guarantee, and every tail recursive call will grow the stack, quickly leading to a stack overflow. In our EVM application, disabling TCO causes a stack overflow, resulting in the application crashing. Additionally, supporting TCO enables Fides to support applications that were explicitly programmed with the intention of utilising TCO.

To support tail-calls, the SM has to identify whether the call preceding the cross-compartment tail-call was a cross-compartment call or an intra-compartment call. As mentioned in §4.1.1, for any inter-compartment call, we replace the return address saved on the program stack with a special address μ outside of the program text. Given this behaviour, at a cross-compartment tail call, if the current return address is μ, then the previous call is a cross-compartment call. This corresponds to $process\_action$: **CP1** $\xrightarrow{call}$ $init\_election$: **CP2** $\xrightarrow{tailcall}$ $reseed\_rng$: **CP3**. The return path must pop the SM context inserted for `init_election` and return to `process_action`. In this case, we do not push **CP2**'s context into the SM stack, and naturally, on return, the call returns to `process_action`.

If the current return address before the tail call is not μ, then the prior call is an intra-compartment call. If so, then nothing special needs to be done by the SM.

4.6 Supporting exception handling

When an exception propagates across compartments, Fides must ensure that the SM stack is unwound consistently with the program's handler chain so that the compartment *context* restored at the handler is correct. OCaml maintains the exception handler chain as a linked list of exception-handler stack (trap) frames. Each trap frame contains the PC of the exception handler and a pointer to the next trap frame. When an exception is raised, control is transferred to the exception handler PC in the head of the trap frame list, and the head element is popped.

In Fides, whenever we cross a compartment boundary, we save the current exception handler program counter in the SM stack along with the compartment context. Importantly, like tail calls, on cross-compartment calls, we update the exception handler PC in the trap frame to a special address ν outside of the executable code region. This ensures that when an exception is raised across compartments, the control traps to the SM. Here, we unwind the SM stack, restore the correct exception handler PC, and transfer control to the handler.

5 Fides Implementation

We realise Fides in the Shakti open-source RISC-V processor [21]. Currently, Fides supports OCaml and C and is tested on Mirage Unikernels[47]. Mirage is a clean-slate Unikernel containing a mix of OCaml and C code bases, making it suitable for evaluating Fides. We extend the MirageOS backend to execute on baremetal RISC-V [34] processors. In this section, we explain the changes made to the hardware and the software stack to support Fides.

5.1 Hardware Changes

Code compartments. To support code compartments, we add one custom instruction, `checkcap`, to the RISC-V ISA. This instruction should be present at all valid compartment entry and return points. The `checkcap` instruction has one argument that indicates the compartment identifier. On a cross-compartment call, this compartment id is passed to the Security Monitor (SM).

Processor pipeline. Custom Control and Status Registers (CSRs) are added to track the current and fluid compartment boundaries. Fides allows (any single or group of adjacent) compartments to be treated as a fluid compartment at runtime by setting the corresponding CSR bits. All these CSRs are protected from direct access

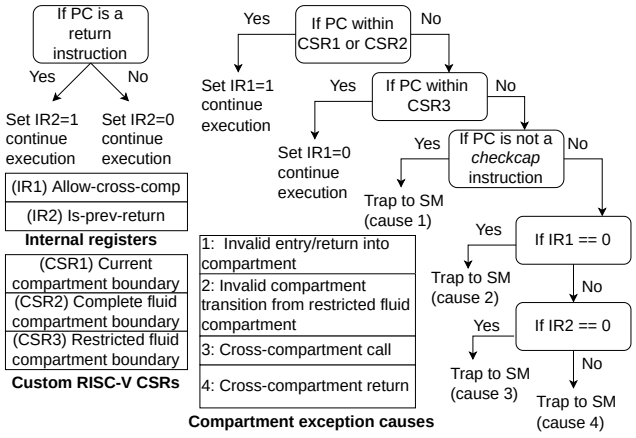

Figure 6: Runtime view of a cross-compartment switch. Checks are introduced in the RISC-V pipeline to ensure control flow does not cross compartment boundaries and traps to SM if it crosses.

to the application code. Access is enabled automatically by the hardware when it traps to the SM and is disabled upon return from the SM. This allows Fides to support compartments for applications running in machine mode (M-mode) on baremetal systems.

The core pipeline is modified to detect cross-compartment calls and returns. The compartment checks incorporated into the pipeline are shown in Figure 6. Whenever a new *pc* is loaded, the hardware checks if it is within the current compartment boundary or within the fluid compartment boundaries (if enabled). If this is not the case, then the hardware traps to the SM with the specific trap code. There are four possible causes, as shown in Figure 6, out of which two denote success, and two denote failure: not landing at a `checkcap` instruction or trying to cross compartments from a restrictive fluid compartment. To differentiate a cross-compartment call from a cross-compartment return, the hardware marks all the return instructions (using an internal state variable), based on the RISC-V ABI [33] and notifies the SM whether the previous instruction that triggered the compartment switch is a return instruction or not. The SM saves or restores the compartment context based on this information.

Data compartments. To support data compartments in C, hardware-assisted fat pointers are used. The hardware and ISA are extended to support custom instructions: `val`. The `val` instruction, which is inserted before every pointer dereference, enforces spatial and temporal memory safety.

5.2 Software Changes

Code compartments. The OCaml and LLVM compilers' RISC-V backends are modified to parse the `.cap` files, which hold the mapping of functions to compartments. The compiler instruments all valid function entry points with the `checkcap` instruction and places all functions belonging to a compartment in the same code section in the ELF generated. The SM initialises each compartment's code region's base and bound at boot time.

Fides' Security Monitor(SM) is isolated and placed outside the bounds of all compartments. The key task of the SM is to enforce the compartment access policy on all cross-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.

To support tail calls across functions in the presence of code compartments, the SM changes the return address, which is to be stored on the program call stack, to a constant magic value during a cross-compartment call. This interferes with the OCaml runtime's garbage collection (GC) stack scanning procedure. To ensure proper functioning, the OCaml runtime and SM share a shadow stack into which a copy of the actual return addresses during cross-compartment calls is pushed. When the GC stack scanning procedure encounters a frame corresponding to the constant magic value, it uses the shadow stack to find the correct return address.

Data compartments. We modify the LLVM compiler's RISC-V backend to introduce a custom fat-pointer transformation pass, which instruments the C code with fat-pointer checks. The memory safety instrumentation does not automatically handle inline assembly. To overcome this issue, we manually modify the inline assembly code to be aware of fat-pointers and insert checkcap instructions at the function entry. Notably, the cost to do this is directly proportional to the amount of inline assembly present, which is expected to be small in real-world C libraries.

Fides' memory-safety instrumentation imposes some restrictions. Currently, we do not support variadic arguments. Fides assumes that the upper bits of the pointer do not contain any application-specific data. The OCaml runtime, implemented in C, is part of the trusted computing base (TCB) and is not compiled with the fat pointer instrumentation. The Fides C 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.

5.3 Engineering effort

Fides extends LLVM version 11.1 and OCaml version 4.14.2 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 in the presence of code compartments. 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 fat pointer instrumentation pass in LLVM, which we build upon, consists of 500 LoC. Observe that implementing Fides only requires minimal self-contained additions to the compilers.

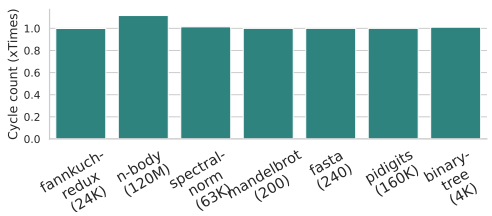

Figure 7: Execution time (clock cycles) of code compartments w.r.t C baseline.

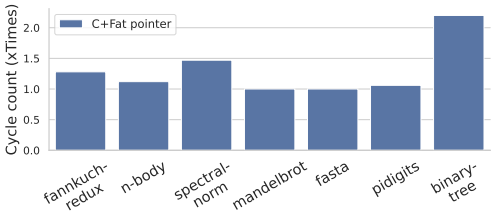

Figure 8: Microbenchmarks execution time (clock cycles) of C secured with fat pointer w.r.t C baseline.

6 Results

6.1 Hardware Overheads

Fides is realized on a Xilinx Artix-7 AC701 FPGA [73] with a default synthesis strategy. The baseline RISC-V core [21] 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 remains unaffected by any of the modifications introduced by Fides. Comparing Fides to CHERIoT [3], CHERIoT uses a tagged architecture and has over a 100% increase in area, while FIDES has a 6% increase.

6.2 Microbenchmark

To quantify the overheads of compartmentalising 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 §4.4. 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. Whereas, placing `Array.iteri` 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 HoF is far less than the overhead of saving and restoring the compartment context. Given that HoFs 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 §4.4, fluid compartments 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 hardware+compiler-assisted fat pointers? Figure 7 shows the clock cycle overhead of enabling code compartments, and Figure 8 shows

!t

Table 3: EVM case study with different compartment (comp) strategies. The baseline is the EVM application without Fides.

# Comp.	Overhead (cycles)	Avg code size / comp (KB)	# Inter-comp trans ($\times 10^4$)
6:Fluid	1.04×	132	5
23:Fluid	1.06×	47	8
6:Non-Fluid	1.19×	132	5170
23:Non-Fluid	1.21×	47	5320

the overhead of the fat pointer scheme with respect to the C baseline. All the microbenchmarks are taken from the Computer Language Benchmarks Game [67]. mandelbrot and fasta are I/O intensive benchmarks, with mandelbrot containing negligible pointer operations. pidigits relies on the GMP library, which was also compiled with memory safety checks.

6.3.1 **Cost of supporting code compartments.** We placed commonly invoked functions in different compartments to understand the overhead of enabling code compartments. We can see that the overhead of compartments is low compared to the microbenchmark in the previous section. On average, there is only a 1.9% increase in execution time when code compartments are enabled compared to when compartments are not enabled. Overall, we can see that the overhead Fides code compartments are low. The SM stack is used to save and restore the compartment metadata. We also save and restore the callee-saved registers (as we do not trust the callee to preserve the registers). This adds up to 160 bytes, including the metadata, saved for inter-compartment calls.

6.3.2 **Cost of hardware-enforced fat pointers.** Figure 8 shows the overhead of C hardened with fat pointers. On average, there is an overhead of 36%. The outlier is the Binary tree program.

The majority of the overhead results from the insertion of the `val` instructions before pointer dereference. If we are able to statically prove at compile time [29] that a pointer dereference is spatially and temporally memory safe, we can safely remove the `val` instruction, eliminating the need for a runtime check. The overheads can further be reduced by employing these static analysis techniques.

6.3.3 **Code size impact.** Fides instruments the program with two new instructions – checkcap and val. We observed that the introduction of new instructions has a minimal impact on code size. The code size increase is 4% on the benchmark programs.

6.4 Evaluating the EVM application

Our technique scales to real-world applications, utilising significant third-party libraries. The EVM application is constructed using 20 existing third-party packages from the MirageOS ecosystem, including mirage-crypto, lwt, etc. In total, it has 68k lines of code (LoC), out of which we wrote 5k lines of new code. 48% of the codebase is in OCaml. MirageOS itself depends on 29K LoC C code, the majority of which is the OCaml runtime (21K LoC). We place the core OCaml runtime in a complete fluid compartment. Commonly used functions like `strcpy`, are placed in a restricted fluid compartment, sandboxing them completely. Access to device-specific functions like `printf`, are restricted to only the required compartments. 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 specify the compartment entry points and their corresponding compartment IDs.

Some OCaml libraries do use `unsafe_*` functions. In the EVM application, seven libraries used unsafe features. While the `unsafe_*` functions could be replaced by their safe counterparts, in our EVM application, we did not restrict the use of these functions. Instead, we manually audited the unsafe features for correctness. We are able to support vast majority of C code out of the box. There were minor uses of inline assembly in the device drivers (30 LoC), which, required manual instrumentation.

We evaluate the overheads of the EVM application with six compartments described in §4. 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 3 presents the results. Compared to the insecure baseline, Fides EVM application with *6 compartments:Fluid* has only 4% overhead in the case of fluid compartments enabled. Without fluid compartments (*6:Non-fluid*), the number of inter-compartment transitions increases significantly, which has a corresponding performance drop, 19% overhead compared to insecure baseline. When the application is compartmentalised in a fine-grained manner (*23 compartments:Fluid*), 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 largely unchanged. This illustrates that if a security engineer puts in the effort to compartmentalise 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 compartmentalisation. The results also show that fluid compartments play a significant role in keeping the overheads low and providing better security by avoiding the confused deputy attack.

6.5 Evaluating HTTPS Webserver

We study securing an HTTPS web server MirageOS unikernel using Fides. In total, the web server is constructed using 54 packages, including the GMP C library. We compartmentalise an HTTPS web server based on 3 different strategies, as shown in Figure 9.

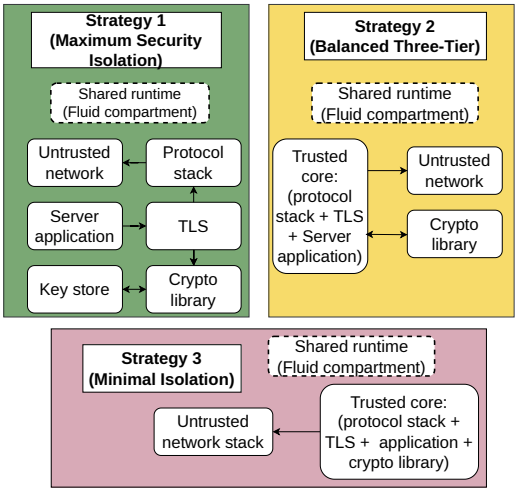

Figure 9: Different strategies for compartmentalising an HTTPS web server. The edges represent the allowed transitions between the compartments.

Strategy 1: 6 compartments → We place the (i) low-level network stack, (ii) protocol layer(TCP/UDP), (iii) server application logic, (iv) TLS layer, (v) crypto library (vi) key-value memory store, each in their own compartments.

Strategy 2: 3 compartments → We adopt a balanced approach, where we split the unikernel into 3 compartments: (i) network stack, (ii) crypto libraries and (iii) all the remaining, protocol + TLS + server logic in 3 different compartments.

Strategy 3: 2 compartments → This is the least secure of all approaches, as it splits the unikernel into just two compartments, the untrusted network stack and all the remaining assigned to a different compartment.

The entire web server, including all third-party packages, is 500K LoC. The .cap files and flags in the build scripts, 250 LoC. The average code size per compartment w.r.t each compartmentalisation strategy: `Strategy 1` resulted in an average of 160KB of code/compartment, `Strategy 2`: 300 KB of code/compartment, and `Strategy 3` resulted in the highest, at 490 KB of code/compartment. The lower the code/compartment, the less the attack surface available for exploitation by an attacker. `Strategy 1` has the least code/compartment among the three.

We measured the throughput of each compartmentalisation scheme under continuous load for 10 minutes. The overall throughput is expected to be low for the baseline core, due to the sub-100 MHz clock frequency and the use of a blocking network device driver. The insecure baseline completed 57 requests, while `Strategy 1` (6 compartments, maximum isolation) handled 46, `Strategy 2` handled 49, and `Strategy 3` (minimal isolation, 2 compartments) completed 52. The capacity loss is inversely proportional to the security isolation level: `Strategy 1` suffers a 19.3% throughput reduction, while `Strategy 3` loses only 8.8%. This demonstrates the

Table 4: Summary of hardware-assisted compartment solutions.

Technique	F1	F2	F3	F4	F5
Secage[44]	512	✗	✗	✗	✗
Glamdring[43]	2	✗	✗	✗	✗
GOTEE[24]	2	✗	✓	✗	✗
Donky[63]	∞	✗	✗	✗	✗
Enclosures[23]	∞	✗	✓	✗	✗
PKRU-Safe[37]	2	✗	✗/✓	✗	✗
Galeed[60]	2	✗	✗/✓	✗	✗
CHERI-JNI[9]	$2^{\dagger}$	-	✗/✓	✗/✓	✓
ACES[13]	∞	✓	✗	✗	✗
MINION[35]	∞	✓	✗	✗	✗
CHERIoT[3]	2^{64}	✓	✗	✓	✓
Fides	256‡	✓	✓	✓	✓

✗/✓: partially supported †: support multiple compartments within C
‡: currently, 256 compartments are supported, extendable to 4096
F1: Maximum number of compartments **F2**: Support for resource-constrained baremetal embedded systems **F3**: Support for memory-safe languages **F4**: Support for fine-grained compartments **F5**: Support for direct access to shared data

fundamental security-performance tradeoff in compartmentalised systems. For resource-constrained embedded systems, Strategy 3 offers the best throughput (9% overhead) but weaker isolation, while Strategy 1 provides maximum security at a 19% capacity cost, making it suitable for high-security applications.

7 Related work

Intra-process compartment techniques have been widely studied over the years. Table 4 compares recent solutions with respect to their support for safe languages, applicability in resource-constrained environments, compartment granularity, and data sharing between compartments.

Enforcing compartments. Donky [63], Enclosures [23] tag pages with compartment IDs. Enclosures uses Intel MPK [31]. Donky extends a similar scheme to RISC-V processors. Secage [44] utilises Intel VT-x [69] to enforce isolation between compartments by setting up separate page tables for each compartment. CETIS [72] utilises Intel CET [31] to support two compartments, while Glamdring [43] and GOTEE [24] utilise Intel SGX [32] to achieve the same. Compared to the above works, Fides does not rely on the MMU, making it ideal for baremetal systems. ACES [13] and MINION [35] do not rely on MMU and use ARM MPU [6].

CHERIoT [3] is similar to Fides: it extends the compiler to emit custom instructions to perform security checks and transforms every pointer into architectural capabilities to define compartment regions and enforce isolation between them. Compared to Fides' fat pointers CHERI capabilities are more expressive, and store permission bits (like `rwx`) also, restricting 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, thereby leveraging the safe language guarantees and making it easier and more straightforward to port a mixed-language application to Fides.

Support for safe languages. Galeed [60] and PKRU-Safe [37] utilize Intel-MPK to secure Rust from C/C++ by splitting the application into two domains. They do not support compartments within Rust or C/C++. GOTEE [24] supports compartments in the Go language using Intel SGX. Enclosures [23] provides package-based isolation in Go and Python. All these techniques rely on paging support, restricting their applicability to embedded systems. CHERI-JNI [9] utilizes CHERI capabilities to secure the Java Native Interface [58] but does not support compartments within the Java code. CHERI supports the Rust [64] language, but is yet to be ported to garbage-collected languages like OCaml. Fides extensions to the OCaml compiler are lightweight §5.3 and do not require extensive changes to the compiler backend.

Support for fine-grained compartments. The compartment granularity determines the reduction in attack surface. Intel-MPK supports 16 compartments but requires software multiplexing to support more compartments, which incurs overheads [26, 59]. ACES [13] does not support fine-grained compartments, as the number of regions a compartment can access is limited based on the MPU register count (8-16). This makes MPU-based techniques unsuitable for protecting multiple separated regions. Fides supports fine-grained compartments, which reduces attack surface significantly.

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. GOTEE [24] and Glamdring [43] 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.

Support for memory safety in C. There are many extant works that aim to enforce spatial and temporal memory safety. CCured [55] enforces spatial memory safety by introducing a fat pointer into the language's type system. Delta Pointers [38] and Low-Fat [19] ensure the same by encoding the bounds metadata within the pointer using compact encoding schemes. Softbound-CETS [53] achieves spatial and temporal memory safety by associating every pointer with disjoint bounds and liveness metadata. Checked-C [20, 74] achieves both spatial and temporal memory safety with the same fat pointer size as CCured has. MarkUs [2] and Dieharder [57] enforce temporal memory safety by ensuring that a freed memory region is not immediately reallocated, resulting in significant memory overheads that make them impractical for resource-constrained environments.

8 Conclusion

Fides is a light-weight hardware-assisted compartmentalisation scheme, specifically designed to cater for mixed-language applications involving unsafe C/C++ and safe languages like OCaml. Fides supports tail-calls, exceptions and HoFs, which are common feature in modern languages nowadays. Fides introduces minimal changes to the underlying hardware, making it suitable for deployment in resource-constrained embedded systems. Fides introduces

two new instructions, which are minimal. Furthermore, Fides's extension to C code with fat pointer-based memory safety helps users securely include legacy third-party codebases and encourages an incremental approach to migrating to a memory-safe language. The fine-grained compartments empower the security engineer to express efficient compartment policies.

References

[1] 2011. *Lambda expressions (since C++11).* https://en.cppreference.com/w/cpp/language/lambda

[2] 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. doi:10.1109/SP40000.2020.00058

[3] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO).* 641–653.

[4] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press, Cambridge, UK.

[5] Arm Limited. 2021. Arm Architecture Reference Manual for A-profile architecture (Armv8-A): Top Byte Ignore (TBI). https://developer.arm.com/documentation/ddi0487/ga. See AArch64 address tagging / Top Byte Ignore.

[6] Ying Bai. 2016. *ARM® Memory Protection Unit (MPU).* 951–974. doi:10.1002/9781119058397.ch12

[7] Olivier Benot. 2011. *Fault Attack.* Springer US, Boston, MA, 452–453. doi:10.1007/978-1-4419-5906-5_505

[8] Blackduck. 2025. THE STATE OF EMBEDDED SOFTWARE QUALITY AND SAFETY 2025. https://www.blackduck.com/content/dam/black-duck/en-us/reports/state-of-embedded-software-quality-and-safety.pdf.

[9] 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. doi:10.1145/3093337.3037725

[10] 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/

[11] Catalin Cimpanu. 2019. *Two malicious Python libraries found and removed from PyPI.* https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/

[12] Michael R. Clarkson. 2022. *Type Checking in OCaml.* https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

[13] 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

[14] Cloudflare. 2020. What is the Mirai Botnet? https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/.

[15] Mitre Corporation. 2023. *Common Weakness Enumeration, A Community-Developed List of Software and Hardware Weakness Types.* https://cwe.mitre.org/

[16] Mircea Cosbuc. 2020. *All About Unikernels: Part 1, What They Are, What They Do, and What's New.* https://blog.container-solutions.com/all-about-unikernels-part-1-what-they-are

[17] 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. doi:10.1145/3316482.3326356

[18] Oracle Java Documentation. 2023. *Lambda Expressions.* https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[19] 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

[20] 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. doi:10.1109/SecDev.2018.00015

[21] 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. doi:10.1109/VLSID.2016.130

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, Reading, MA, USA.

[23] 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. doi:10.1145/3445814.3446728

[24] 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.

[25] GoLang 2022. *A guide to the Go Garbage Collector.* https://go.dev/doc/gc-guide

[26] 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

[27] Norm Hardy. 1988. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.* 22, 4 (oct 1988), 36–38. doi:10.1145/54289.871709

[28] 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

[29] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2024. Top of the Heap: Efficient Memory Error Protection of Safe Heap Objects. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24).* Association for Computing Machinery, New York, NY, USA, 1330–1344. doi:10.1145/3658644.3690310

[30] Industrial Cyber. 2025. Forescout's 2025 report reveals surge in device vulnerabilities across IT, IoT, OT, and IoMT. https://industrialcyber.co/reports/forescouts-2025-report-reveals-surge-in-device-vulnerabilities-across-it-iot-ot-and-iomt/.

[31] 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

[32] Intel. 2014. *Intel Software Guard Extensions Programming Reference.* https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf

[33] RISC-V International. 2019. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA.* https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[34] RISC-V International. 2021. *RISC-V International.* https://riscv.org/

[35] 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.*

[36] 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.

[37] 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. doi:10.1145/3492321.3519582

[38] 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. doi:10.1145/3190508.3190553

[39] The Rust Programming Language. 2011. *Rust borrowing.* https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

[40] The Rust Programming Language. 2011. *Rust type system.* https://doc.rust-lang.org/reference/type-system.html

[41] Stefan Lankes, Jens Breitbart, and Simon Pickartz. 2019. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (Huntsville, ON, Canada) *(PLOS'19).* Association for Computing Machinery, New York, NY, USA, 8–15. doi:10.1145/3365137.3365395

[42] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. [n. d.]. The OCaml system: Documentation and user's manual. *INRIA* 3 ([n. d.]), 42.

[43] 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.

[44] 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).

[45] 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. doi:10.1145/3456629

[46] Anil Madhavapeddy and Yaron Minsky. 2022. *Understanding the Garbage Collector.* https://dev.realworldocaml.org/garbage-collector.html Chapter in *Real World OCaml*, 2nd edition.

[47] 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. doi:10.1145/2451116.2451167

[48] Anil Madhavapeddy, David J. Scott, Patrick Ferris, Ryan T. Gibb, and Thomas Gazagnaire. 2025. Functional Networking for Millions of Docker Desktops (Experience Report). *Proc. ACM Program. Lang.* 9, ICFP, Article 256 (2025), 19 pages. doi:10.1145/3747525

[49] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

[50] 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/

[51] Mozilla. 2021. *Mozilla Welcomes the Rust Foundation.* https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation/

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

[53] 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. doi:10.1145/1542476.1542504

[54] NCSC UK. 2021. Log4j vulnerability – what everyone needs to know. https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know.

[55] 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. doi:10.1145/1065887.1065892

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

[57] 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. doi:10.1145/1866307.1866371

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

[59] 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.

[60] 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. doi:10.1145/3485832.3485903

[61] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages. doi:10.1145/2133375.2133377

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

[63] 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.

[64] 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

[65] 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/

[66] 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. doi:10.1109/SP.2013.13

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

[68] 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/

[69] 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. doi:10.1109/MC.2005.163

[70] Jérôme Vouillon. 2008. Lwt: A Cooperative Thread Library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML* (Victoria, BC, Canada) *(ML '08)*. Association for Computing Machinery, New York, NY, USA, 3–12. doi:10.1145/1411304.1411307

[71] 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. doi:10.1109/SP.2015.9

[72] 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. doi:10.1145/3548606.3559344

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

[74] 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. doi:10.1145/3586038

Appendix

A Memory error-based attack

```
module F () = struct
  let admin_flag = ref false
  let is_admin () = !admin_flag
end

module M : sig
  val is_admin : unit -> bool
end = F ()

let _ = Callback.register "is_admin" M.is_admin

let main () =
  Malicious_ml.init();
  if (M.is_admin ()) then print_endline "Leak!"

let _ = main ()
```

Listing 1: `sensitive.ml`

```
external init_c : unit -> unit = "init"
let init () = init_c ()
```

Listing 2: `malicious_ml.ml`

```
#include <caml/mlvalues.h>
#include <caml/callback.h>
#include <caml/memory.h>

value init (value unit) {
  value *is_admin = caml_named_value("is_admin");
  value admin_flag = *is_admin + 32;
  Field(admin_flag, 0) = Val_int(1);
```

```
  return Val_unit;
}
```

Listing 3: malicious_c.c

The attack in malicious_c.c utilises the knowledge of how the module M in sensitive.ml is laid out on the heap and performs an out-of-bound write using the is_admin pointer to update the value of the private admin_flag field to true. The program can be compiled and run as follows:

```
% ocamlopt -g -o leak.exe malicious_ml.ml
    sensitive.ml malicious_c.c
% ./leak.exe
Leak!
```

B Control-flow subversion in the absence of isolation

```
let secret = [| 42; 42 |]
let sum = Sys.opaque_identity (Array.fold_left
    (+) 0)
(* [opaque_identity] prevents inlining and forces
    closure allocation *)

let _ = Callback.register "sum" sum
(* register [sum] as a callback so that C
    functions can call it *)

let main () =
  Malicious_ml.init(); (* some initialisation *)
  sum secret (* leaks! *)
  |> ignore

let () = main ()
```

Listing 4: sensitive.ml

```
let leak arg =
  Array.iter (fun x -> Printf.printf "%d " x) arg;
  Printf.printf "\n%!"

let () = Callback.register "leak" leak

external init_c : unit -> unit = "init"

let init () = init_c ()
```

Listing 5: malicious_ml.ml

```
#include <caml/mlvalues.h>
#include <caml/callback.h>
#include <caml/memory.h>

value init (value unit) {
  value *callback_sum = caml_named_value("sum");
  value *callback_leak = caml_named_value("leak");
  Store_field (*callback_sum, 0,
      Field(*callback_leak, 0));
  /* overwrite the code pointer in the sum
      closure with that of the leak function */
  return Val_unit;
}
```

Listing 6: malicious_c.c

The attack in malicious_c.c utilises the knowledge of the closure layout to overwrite the code pointer in the original function with that of the malicious leak function. Importantly, the write is within the bounds of the sum closure, and hence, spatial memory safety will not prevent this attack. The program can be compiled and run as follows:

```
% ocamlopt -g -o leak.exe malicious_ml.ml
    sensitive.ml malicious_c.c
% ./leak.exe
42 42
```

which leaks the secret. In Fides, one may place the sensitive and malicious modules in separate compartments with the init function tagged as a compartment entry point but not leak. With this, at the call to sum in main, the execution would trap. It is useful to note that preventing this attack requires compartments for the OCaml code. A scheme which places all the OCaml code in the same compartment would not prevent this attack.