Sandboxing is a common technique that allows low-level, untrusted components to safely interact with trusted code. However, previous work has only investigated the low-level memory isolation guarantees of sandboxing, leaving open the question of the end-to-end guarantees that sandboxing affords programmers. In this paper, we fill this gap by showing that sandboxing enables reasoning about the known concept of robust safety, i.e., safety of the trusted code even in the presence of arbitrary untrusted code. To do this, we first present an idealized operational semantics for a language that combines trusted code with untrusted code. Sandboxing is built into our semantics. Then, we prove that safety properties of the trusted code (as enforced through a rich type system) are upheld in the presence of arbitrary untrusted code, so long as all interactions with untrusted code occur at the "any" type (a type inhabited by all values). Finally, to alleviate the burden of having to interact with untrusted code at only the "any" type, we formalize and prove safe several wrappers, which automatically convert values between the "any" type and much richer types. All our results are mechanized in the Coq proof assistant.

CCS Concepts: • Security and privacy → Logic and verification; • Theory of computation → Programming logic.

Additional Key Words and Phrases: Sandboxing, robust safety, Iris, type systems, logical relations, language-based security

ACM Reference Format:

1 INTRODUCTION
The Internet makes it easy to download useful software components from untrusted authors. While it is convenient to integrate such untrusted components into a trusted (i.e., otherwise safe) application, doing so also incurs security risks. The untrusted code could violate the integrity of the trusted application’s private data structures, steal secrets, or take over the system and install malicious software. In all these cases, the untrusted code violates expected safety properties of the trusted application. Thus, it is important to ensure that an application maintains its expected safety properties even when linked and co-executed with arbitrary untrusted code. This is often called...
robust safety [Grumberg and Long 1994; Gordon and Jeffrey 2001; Fournet et al. 2007; Abadi 1997; Bugliesi et al. 2011; Swasey et al. 2017].

One common engineering technique for ensuring secure interoperation between trusted and untrusted code is to physically sandbox the untrusted parts of an application at coarse granularity using hardware, kernel, or library support for isolation. Memory is partitioned into low and high compartments, and the hardware and kernel enforce that untrusted code—sandboxed in the low compartment—cannot directly access the memory in the high compartment (where trusted code operates) even if it can guess private memory addresses of the trusted code [Koning et al. 2017]. Additionally, untrusted code cannot directly access system calls. Examples of such techniques are software fault isolation by rewriting untrusted code [Yee et al. 2009], in-user-space sandboxing of untrusted libraries [Mozilla 2019; Lamowski et al. 2017; Google 2019; Vahldiek-Oberwagner et al. 2019], use of multiple kernel-backed address spaces within an application [Litton et al. 2016], and the use of modern features of CPUs like secure enclaves [McKeen et al. 2013; ARM Limited 2009].

Although sandboxing is widely used, and prior work has shown formally that specific sandboxing techniques attain intrinsic properties like memory isolation, to the best of our knowledge there is no clear understanding of what end-to-end reasoning sandboxing affords programmers. Our goal in this paper is precisely to fill this gap: we show that sandboxing allows programmers to reason about the robust safety of trusted code. As explained above, robust safety is a well-studied concept, which means that the trusted code’s safety properties hold even when co-executing with arbitrary untrusted code. In verification terms, sandboxing allows reasoning about the safety properties of trusted code without having to consider the behavior of untrusted code during verification.

To formalize this, we start by designing an idealized but representative operational semantics with built-in sandboxing. Next, we formally show that, if untrusted code is properly sandboxed, robust safety is an automatic (free) consequence of verifying safety properties of the trusted code without any assumptions about the types of values it receives from untrusted code. Specifically, if we verify a safety property of trusted code assuming that all its interactions with the untrusted code are at a type we call any—which includes all values of the language—then that safety property also holds when the trusted code runs co-linked with any untrusted code.

As an add-on, we extend our formalization to another low-level engineering technique that is commonly used in conjunction with sandboxing. This technique, often called “wrapping” or “marshalling” [Mozilla 2019; Lamowski et al. 2017; Google 2019; Swasey et al. 2017], dynamically converts values between types that two different components of an application use internally. In the context of sandboxing, where we assume nothing of the untrusted code, wrapping can be used to convert between types that the trusted code understands and the type any. This allows the programmer to write typed interfaces between trusted and untrusted code; wrapping automatically takes care of checking and transforming values at runtime so that the actual interactions between trusted and untrusted code are at type any, as required by our first result.

Although wrappers simplify the programmer’s task, they obviously move the onus of security to the correctness of the wrappers. Thus, as a second contribution, we design and verify an (extensible) library of wrappers that convert between the types of a rich type system and the type any. Our wrappers go beyond existing work in two ways. First, our type system includes substructural (affine) features and ownership types to a limited extent, modeling some aspects of modern languages like Rust [The Rust Developers 2019], and we show how dynamic wrapping can handle such features. Second, sandboxing imposes low-level requirements on wrappers—for example, when exporting an object from trusted to untrusted code, a wrapper must also copy the object from the high to the low compartment, as untrusted code cannot access the high compartment.
Relating to existing work. The idea of robust safety originated in the context of modular model checking [Grumberg and Long 1994]. Early security-related work by Gordon and Jeffrey [2001] presents a type system for the spi calculus wherein each program that can be typed at the special Un type is robustly safe against arbitrary untrusted code. Several other papers use similar ideas [Fournet et al. 2007; Abadi 1997; Bugliesi et al. 2011]. More recent work by Swasey et al. [2017] applies ideas from robust safety to the verification of object capability patterns. They use a program logic based on Iris [Jung et al. 2018b] to allow modular reasoning about the trusted code in a language similar to JavaScript. However, all this previous work on robust safety focuses on languages with abstract memory models, where it is possible to hide a secret memory location by simply not sharing it with untrusted code. Sandboxing is not useful in this setting. In contrast, we are interested in robust safety in the presence of a concrete (low-level) memory model where the untrusted code is more powerful in that it can guess or compute memory addresses of the trusted code and dereference them. This justifies the need for sandboxing.

There is also work on verification of sandboxing techniques like software-fault isolation [Besson et al. 2019; Morrisett et al. 2012], architectural meta-data tagging (micro-policies) [de Amorim et al. 2015], and architectural memory capabilities [El-Korashy 2016]. However, these verification efforts are directed at meta-theoretic correctness properties of the sandboxing mechanisms themselves, e.g., that sandboxed code cannot directly access memory outside the sandbox. In contrast, our work establishes how sandboxing contributes to the robust verification of an application that uses the sandbox. In fact, our operational semantics builds in a correct, abstract sandbox, and all our work is done relative to this operational semantics.

Wrappers are a widely deployed software-engineering tool. They are primarily used to reduce development effort and improve software maintainability, but in some applications, notably Mozilla’s Firefox [Mozilla 2019], wrappers are also used to improve security. Wrappers are often combined with sandboxing in practice, e.g., in Lamowski et al. [2017], but they are typically limited to marshalling and unmarshalling records of first-order values. In contrast, our wrappers cover a rich type system with affine and ownership types, and we show how these types can be wrapped and unwrapped securely. The design of our wrappers is inspired by work on higher-order contracts [Findler and Felleisen 2002; Blume and McAllester 2004]. Prior work like that of Swasey et al. [2017] has verified several common wrappers but in an abstract memory model. Our wrappers are instead verified in a low-level memory model in the context of sandboxing, which results in very different proof requirements.

Summary of contributions and technical work. We give the first formal account of the high-level (end-to-end) benefits of low-level sandboxing. We apply the idea of robust safety to sandboxing in the setting of a concrete (low-level) memory model where addresses can be guessed or computed a priori, thus obtaining strong end-to-end guarantees about the behavior of trusted code when linked with arbitrary sandboxed code. Additionally, through a library of verified wrappers, we show how several common types (including affine and ownership types) can be wrapped and unwrapped at sandbox boundaries, thus allowing seamless integration of untrusted code into the verification of a trusted application.

Our technical contributions are as follows:

- We idealize and formalize memory and system call isolation using a small core calculus which builds in two memory partitions and two compartments (one trusted and one untrusted) – this is the simplest but also one of the most common settings for sandboxing. Section 3 introduces this language (called $\lambda_{\text{sandbox}}$) and its operational semantics.
- After introducing a rich affine type system for $\lambda_{\text{sandbox}}$ in Section 4, we show formally that sandboxing yields a principle for robust verification of safety properties of a typed $\lambda_{\text{sandbox}}$.
In this section we introduce the key ideas of our work using two examples of concrete (trusted) applications that interact with sandboxed untrusted code.

Unsafe printing. Our first example is contrived but illustrative. Assume that the operating system provides a system call unsafe_print that prints its numerical argument on the screen. This system call, however, has one fatal flaw: When called with a number with more than three decimal digits, it will overflow an internal buffer of the kernel. Thus, the security goal—a safety property—is to ensure that unsafe_print is never called with an argument greater than 999.

A reader familiar with the usual definition of safety in the programming languages literature might expect an additional condition that the program must not get stuck in a small-step operational semantics. However, it is not possible to fulfill this condition in our setting where programs may include arbitrary untrusted code as such code could cause stuckness trivially by crashing the program. Consequently, standard program safety (non-stuckness) is not what we are after. Instead, we formalize our safety properties (only) on traces of externally visible behavior of the program, i.e., the sequences of system calls executed by the program. In our example, we will consider a program safe as long as any trace of system calls it creates does not include any unsafe_print with an argument greater than 999.

Consider two applications in Figure 1 calling unsafe_print. The first application, main, is clearly safe since the argument to unsafe_print is a value read from a location l that previously had 42 written to it. The other application, mainsandbox, is similar to main except that it also runs a function untrusted_main concurrently (treat the primitive gated_call simply as a call instruction in the language for now). Importantly, untrusted_main is untrusted meaning that it can be arbitrary and might try to violate the safety property outlined above. In particular, untrusted_main could
malicious untrusted code

\[
\text{untrusted}\_\text{main}_1 \triangleq \lambda \_. \ \text{syscall}(\text{unsafe\_print}, 1000) \\
\text{untrusted}\_\text{main}_2 \triangleq \lambda \_. \ \text{let} \ l = \text{guess\_location}(); \ l \leftarrow 1000
\]

Fig. 2. Malicious instantiations of the untrusted code.

directly invoke \text{unsafe\_print} with an argument greater than 999, as illustrated by the instance \text{untrusted}\_\text{main}_1 in Figure 2. Alternatively, it could guess the address of the location \( l \) and write a value greater than 999 to it, causing the \text{unsafe\_print} in the body of \text{main}_{\text{sandbox}} to be called with an argument greater than 999. This is illustrated by the instance \text{untrusted}\_\text{main}_2 in Figure 2. In either case, the safety property is violated. Note that the second of these violations is only possible in a low-level language, where references are computable. We deliberately choose this setting as it is more challenging than a language with abstract references and is also closer to systems programming languages like C.

The idea behind robust safety, as proposed in prior work, is to ensure that \text{main}_{\text{sandbox}} maintains its safety property no matter what the untrusted function \text{untrusted}\_\text{main} is. One way to attain this property is to sandbox the execution of \text{untrusted}\_\text{main}, i.e., to contain or limit its effects. The primitive \text{gated\_call} that wraps \text{untrusted\_main} in Figure 1 in fact models such a sandbox in our calculus. The details of the implementation of the sandbox are irrelevant for our purposes – several existing techniques were outlined in Section 1. Instead, we are interested in the following two more foundational questions: (1) What properties are required of the sandbox to make \text{main}_{\text{sandbox}} robustly safe, i.e., safe against any \text{untrusted}\_\text{main} function provided by the untrusted code? (2) Given these properties of the sandbox, how can we prove that \text{main}_{\text{sandbox}} is robustly safe?

For the first question, it should be clear from the examples of Figure 2, that the sandbox must provide at least two properties. (a) Memory isolation: The sandbox must prevent code in it from accessing memory locations private to code outside the sandbox. This prevents the safety violation with \text{untrusted}\_\text{main}_2 by preventing this function from writing \( l \). (b) System call isolation: To rule out attacks like that in \text{untrusted}\_\text{main}_1, the sandbox must prevent code running inside it from making system calls that could violate safety properties of interest. Since we would like to be able to enforce any safety property using the sandbox, it must, in fact, prevent code inside it from making any system call. There is still the question of whether the sandbox needs yet another property to ensure robust safety. In later sections, we answer this question in the negative for a general programming model with system calls and computable memory addresses. Thus, system call isolation and memory isolation suffice for robust safety even in a low-level setting.\(^1\)

For the second question, we show in Section 5 that the use of a sandbox yields a strong principle for verification of robust safety: if a safety property of trusted code is verified without making any assumptions about the values it receives from untrusted code, then the safety property in fact holds robustly. We formalize this principle by introducing a type system for verifying safety properties of trusted code and including a universal type \text{any} containing all values at which trusted code can interact with untrusted code.

\(^1\)Prior work [Gordon and Jeffrey 2001; Fournet et al. 2007; Abadi 1997; Swasey et al. 2017] has established similar principles for robust safety in high-level languages with abstract (non-computable) references. Our work extends those results to low-level languages with computable references. The difference is substantial: with computable references, memory isolation must be provided by the sandbox.
want to give untrusted code access to system calls, but perhaps with some checks. One way to do this is for the trusted code to expose to untrusted code an API that wraps system calls to make sure that the system calls do not violate safety properties. For instance, in our example, trusted code may expose the following function `checked_print` to untrusted code.

```
checked_print ≜ λ x. if x < 1000 then syscall(unsafe_print, x) else stuck
```

This function calls `unsafe_print` with its argument but only if the argument is less than 1000. As a result, it does not violate our safety property, no matter what argument is passed to it. Consequently, this function, which is just a wrapper around `unsafe_print`, can be shared with untrusted code.

While it is possible to write such wrappers manually in principle, this can quickly become unmanageable and buggy for large interfaces. It would be much less error-prone to write an API function that describes the check \( x < 1000 \) using a type. For instance, suppose we define the type `LessThan1000 ≜ \{ z \mid z < 1000 \}`. Then, a programmer could instead write the following function:

```
typed_print ≜ λ x : LessThan1000. syscall(unsafe_print, x)
```

This function is much easier to write, less likely to have a bug (as it does not check its argument in code) and conveys the same intent as the function `checked_print`. The problem with this function is that it maintains our safety property only when its caller passes it an argument that really is of type `LessThan1000`, i.e., an argument that really is less than 1000. Since the caller here is untrusted code, this is not guaranteed.

Ideally, we would like an automatic mechanism that converts the function `typed_print`, whose argument is statically typed, into the function `checked_print`, whose argument may be anything (of the universal type) and which checks the expected type dynamically. This idea, which is quite similar to runtime type checking in gradual and hybrid types [Flanagan 2006; Lehmann and Tanter 2017], is often implemented in practice through a set of wrappers. A wrapper takes a function with a typed interface (like `typed_print`) and converts it to a function whose argument and return types are all universal (like `checked_print`), by inserting dynamic type-checks around the original function.

Under the hood, wrappers rely on a type-indexed family of coercion functions `τ.import : any → τ` and `τ.export : τ → any` to check the result and the arguments of the wrapped function being exported from trusted to untrusted code. In particular, we can define the following macro that implements a simple, generic wrapper:

```
wrap (\( \nu : \tau \rightarrow \tau' \)) ≜ \lambda x. (\( \tau'.export(\( \nu.\tau.import(x)) \)) : any → any
```

Each type can provide its own import and export functions. For example, import for `LessThan1000` simply checks whether its argument is below 1000:

```
LessThan1000.import ≜ \lambda x. if x < 1000 then x else stuck : any → LessThan1000
```

However in the presence of sandboxing, the import/export function of a type may have to do more work than merely checking values. For example, the export function may have to copy the value from the trusted code’s private memory, which is inaccessible to sandboxed untrusted code due to memory isolation, to memory accessible in the sandbox. Further, as we show in the next example, wrappers around affine types are particularly interesting. Such types are checked dynamically by maintaining local state within the import/export functions. As a result, import/export functions can be quite complex and it is imperative to verify their correctness. In Section 6, we describe such verified import/export functions for a rich collection of types.

---

2The notation and idea for this type are inspired by work on refinement types [Freeman and Pfenning 1991; Xi and Pfenning 1998].
Wrappers with affine types. We now look at an example with a more interesting safety property. Suppose our operating system offers three system calls to manipulate files – open, close, and read. The open call returns a file handle (an integer), while close and read take a previously opened handle as argument. Importantly, if close or read are passed an invalid file handle – one that has either not been obtained from a previous open or one on which close has been called earlier – then they can cause the operating system to crash. Consequently, our safety property is that the file handle passed to read or close must have been obtained previously through open without an intervening close on the handle. This safety property can be succinctly represented as a state machine shown in Figure 3. The variable $z_f$ refers to a file handle and the universal quantification at the beginning simply means that the property is really a product of the state machines for all possible file handles.

As in the previous example, the trusted code would like to expose an API to the untrusted code mirroring these three system calls, while ensuring robust safety. The question is what the types of the API functions, say do_open, do_close, and do_read, should be. Since one of the concerns is that read or close may be called with a file handle that was not obtained through open, an obvious thing is to replace the concrete integer file handles with an abstract type file, whose instances can only be obtained via do_open. The API can maintain a bijective association between instances of file and integer file handles using a private table. However, this alone does not suffice. Additionally, the API must ensure that a handle on which do_close has been called is not accidentally reused. For this, the API can declare the type file affine. The affineness can be enforced by removing the entry for the passed handle from the private table in the import function of the file type and inserting it in the export function. The resulting specification of the API (as written by the programmer) is shown in Figure 4, and the implementation of file.import and file.export is shown in Figure 5. The actual functions shared with untrusted code are obtained by applying wrap to the three functions shown in Figure 4.

This implementation of file.import and file.export is inspired by a more general, widely-used programming pattern called dynamic sealing [Morris 1973]. Prior work [Swasey et al. 2017] verified the standard dynamic sealing pattern but to the best of our knowledge our version of dynamic sealing with affine types is new. We verify this pattern in Section 6.
\[ m \] is a global table that maps instances of file to integer file handles:

\[
\text{file.import} \triangleq \lambda x. \begin{cases} \text{get_and_remove}(m, x) & \text{if } x \in m \\ \text{stuck} : \text{any} \rightarrow \text{file} \end{cases}
\]

\[
\text{file.export} \triangleq \lambda x. \begin{cases} \text{fresh_key}(m) & \Rightarrow k \\ \text{put}(m, k, x) & \Rightarrow k : \text{file} \rightarrow \text{any} \end{cases}
\]

Fig. 5. Wrappers for file.

\[ z \in \mathbb{Z} \]
\[ s \in \text{Name} \]
\[ \text{Priv} \ni p ::= \text{Low} | \text{High} \]
\[ \text{Loc} \ni l ::= (p, z_b, z_o) \]
\[ \text{Lit} \ni q ::= \mathbb{N} | z | l \]
\[ \text{Val} \ni v ::= q | \mathit{rec}(x) = e \]
\[ \text{Expr} \ni e ::= x | v | e + e | e - e | e \leq e | e = e | e, e | e(\bar{e}) | !e | e \leftarrow e \]
\[ \text{CAS}(e_0, e_1, e_2) | \text{alloc}(e) | \text{free}(e_0, e_1) | \text{case } e \text{ of } \bar{e} | \text{fork } e \]
\[ \text{mk_location}(e_p, e_b, e_o) | \text{get_priv}(e) | \text{get_block}(e) | \text{get_offset}(e) \]
\[ \text{syscall}(s, e) | \text{gated_call}(s, \bar{e}) | \text{gated_ret}(e) \]

\[ \text{SurfaceExpr} \ni e ::= \{ e \in \text{Expr} \mid e \text{ does not contain } \text{gated_ret} \} \]
\[ \text{SurfaceVal} \ni v ::= \{ v \in \text{Val} \mid v \text{ does not contain } \text{gated_ret} \} \]
\[ \text{Ectx} \ni K ::= (\text{standard left-to-right evaluation contexts}) \]

Fig. 6. Definition of \( \lambda_{\text{sandbox}} \).

\[ \lambda x. e \triangleq \text{rec } (\bar{x}) = e \]
\[ e_1; e_2 \triangleq \text{let } = e_1; e_2 \]
\[ \text{stuck} \triangleq 0(0) \]

Fig. 7. Standard derived forms.

3 LANGUAGE AND OPERATIONAL SEMANTICS

\underline{Notation.} We use overline notation to denote a list (vector) of elements. If a statement contains multiple overlines, they all represent lists of the same length.

Figure 6 shows the syntax of \( \lambda_{\text{sandbox}} \), which at its core is a standard concurrent lambda calculus modeled after \( \lambda_{\text{Rust}} \) [Jung et al. 2018a]. Standard abbreviations are displayed in Figure 7. \( \lambda_{\text{sandbox}} \) features standard binary operations and functions with multiple arguments. The heap is block-based like in \( \lambda_{\text{Rust}} \) and CompCert [Leroy et al. 2012]. Each location contains an integer block identifier \( z_b \) and an integer offset in the block \( z_o \). The expression \( \text{alloc}(z) \) allocates a fresh block of size \( z \) initialized to the poison value \( \mathbb{N} \), \( \text{free}(z, l) \) frees the block at location \( l \) of size \( z \), and \( l.z \) adds \( z \) to the offset of the location \( l \).

\[^3\text{As in } \lambda_{\text{Rust}}, \mathbb{N} \text{ is a special value for uninitialized memory. All operations except storing and reading a poison value to and from memory get stuck.}\]

\[ h \in \text{Heap} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Val} \quad o \in \text{Observation} \triangleq \text{Name, Lit, Lit} \quad \pi \in \text{Trace} \triangleq \text{list Observation} \]

\[ I \in \text{Interface} \triangleq (\text{Priv}, \text{Name}) \xrightarrow{\text{fin}} \text{Val} \quad A \in \text{SyscallAssumptions} \triangleq \text{Trace} \rightarrow \text{Prop} \]

\[ p \subseteq p \quad \text{Low} \subseteq \text{High} \]

\[ h(l) = v \]

\[ I \vdash h, \pi \mid \mathcal{L} \mathcal{O} \mathcal{R} \mathcal{E} \mathcal{A} \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{S} \]

\[ O\text{-write} \quad h(l) = v' \quad l = (p', z_b, z_o) \quad p' \subseteq p \]

\[ I \vdash h, \pi \mid \mathcal{L} \mathcal{O} \mathcal{R} \mathcal{E} \mathcal{A} \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{S} \]

\[ O\text{-syscall} \quad A(\pi \mathcal{A}(s, q_a, q_r)) \]

\[ I \vdash h, \pi \mid \text{syscall}(s, q_a) \rightarrow A h \mid \pi \mathcal{A} \]

\[ O\text{-gated-call} \quad \mathcal{L} \mathcal{O} \mathcal{R} \mathcal{E} \mathcal{A} \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{S} \]

\[ p = \text{High} \Leftrightarrow p' = \text{Low} \quad I((p', s)) = v \]

\[ I \vdash h, \pi \mid \text{gated call}(s, \overline{v}) \rightarrow A h \mid \pi \mathcal{A} \]

\[ O\text{-gated-ret} \quad p = \text{High} \Leftrightarrow p' = \text{Low} \]

\[ I \vdash h, \pi \mid \text{gated ret}(v (\overline{v})) \rightarrow A h \mid \pi \mathcal{A} \]

\[ O\text{-mk-location} \quad p' = \text{Low} \Leftrightarrow z_p = 0 \quad z_b \geq 0 \quad l = (p', z_b, z_o) \]

\[ I \vdash h, \pi \mid \text{mk location}(z_p, z_b, z_o) \rightarrow A h \mid \pi \mathcal{L} l \]

\[ O\text{-get-priv} \quad l = (p', z_b, z_o) \quad p' = \text{Low} \Rightarrow z_p = 0 \quad p' = \text{High} \Rightarrow z_p = 1 \]

\[ I \vdash h, \pi \mid \text{get priv}(l) \rightarrow A h \mid \pi \mathcal{L} z_p \]

\[ O\text{-get-block} \quad l = (p', z_b, z_o) \]

\[ I \vdash h, \pi \mid \text{get block}(l) \rightarrow A h \mid \pi \mathcal{L} z_b \]

\[ O\text{-get-offset} \quad l = (p', z_b, z_o) \]

\[ I \vdash h, \pi \mid \text{get offset}(l) \rightarrow A h \mid \pi \mathcal{L} z_o \]

\[ I \vdash h, \pi \mid E \rightarrow A h', \pi' \mid E' \]

\[ I \vdash h, \pi \mid \mathcal{L} \mathcal{O} \mathcal{R} \mathcal{E} \mathcal{A} \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{S} \]

\[ \mathcal{L} \mathcal{O} \mathcal{R} \mathcal{E} \mathcal{A} \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{S} \]

\[ I \vdash h, \pi \mid e \rightarrow A h', \pi' \mid e'_1 \rightarrow A h', \pi'_1 \mid e'_2 \rightarrow A h', \pi'_2 \]

\[ I \vdash h, \pi \mid E \mathcal{R} [K[e] \rightarrow A h', \pi' \mid E' \mathcal{R} [K[e'_1] \rightarrow A h', \pi'_1 \mid E' \mathcal{R} [K[e'_2] \rightarrow A h', \pi'_2 \]

Fig. 8. Selected rules from the operational semantics of \( \lambda_{\text{sandbox}} \).

As motivated in the previous sections, we want to model strong adversaries that can guess arbitrary locations. Thus, \( \lambda_{\text{sandbox}} \) provides the \texttt{mk\_location}, \texttt{get\_priv}, \texttt{get\_block}, and \texttt{get\_offset} instructions, which allow composing/splitting a location from/into its parts. Using \texttt{mk\_location}, it is possible to implement the guess\_location function from Section 2 by forging a high location from arbitrary block and offset integers.

The non-standard rules of the thread reduction relation

\[
I \vdash h, \pi \mid e \texttt{at } p \rightarrow_A h', \pi' \mid e'_1 \texttt{at } p'_1, (e'_2 \texttt{at } p'_2)^2
\]

are displayed in Figure 8.\(^4\) The elements of this judgment are explained below in more depth, but at a high level it states that executing an expression \( e \) at privilege level \( p \) with the state consisting of the heap \( h \) and the trace of already-executed system calls \( \pi \) can result (after one step) in \( e'_1 \) at privilege level \( p'_1 \) with potentially modified heap \( h' \) and trace \( \pi' \). Only the rule for \texttt{fork} uses the optional second resulting expression \((e'_2 \texttt{at } p'_2)^2\) to signify that it spawned a new thread. The functions that represent the interface between the trusted and untrusted code are contained in \( I \).

The main reason for including concurrency in \( \lambda_{\text{sandbox}} \) is to more faithfully model practical isolation techniques, which typically operate in a concurrent setting [Mozilla 2019; Lamowski et al. 2017; Google 2019; Litton et al. 2016]. Consequently, as we will see in Section 6, we will need to verify that wrappers handle concurrent accesses correctly.

\textit{Privilege level.} To differentiate code running inside the sandbox from code running outside it, an expression \( e \) is always executed at a thread-local privilege level \( p \). Specifically, \( e \texttt{at } \text{High} \) states that \( e \) is executing outside of the sandbox. Only the trusted code should run at \text{High} privilege. Likewise, \( e \texttt{at } \text{Low} \) signifies that \( e \) runs inside the sandbox; untrusted code should always run at \text{Low} privilege.

To switch the privilege level, \( \lambda_{\text{sandbox}} \) provides a \textit{call gate} mechanism, often employed by real-world sandboxing techniques [Lamowski et al. 2017; Google 2019; Litton et al. 2016; Vahldiek-Oberwagner et al. 2019]. Namely, a switch of the privilege is combined with a call to a function preregistered by the code on the other side of the sandbox. Such a mechanism also ensures well-bracketing of the switches, \textit{i.e.}, the call always returns to the code that initiated it and the privilege is automatically switched back. This abstraction of a call gate is modeled in \( \lambda_{\text{sandbox}} \) using the \texttt{gated\_call} and \texttt{gated\_ret} instructions. The preregistered functions are stored in the \textit{interface}

\[
I : (\text{Priv, Name}) \xrightarrow{\text{fin}} \text{Val}
\]

This interface \( I \) cannot be modified during run time but is fixed with the initial state of the program. As can be seen in O-GATED-CALL, \texttt{gated\_call} switches the privilege from \( p \) to the opposite privilege \( p' \) and calls the function registered under the same \( s \) in the interface of the target privilege \( p' \). The application \( v (\overline{t}) \) is wrapped into a \texttt{gated\_ret}, which ensures that the privilege is switched back to the original privilege \( p \) once the called function finishes executing. The \texttt{gated\_call} instruction is bidirectional: it can be used either to enter the sandbox or to leave it.

The \texttt{gated\_ret} instruction is an artifact of ensuring well-bracketing of the privilege switching and thus is not a part of the surface language, existing only during execution of a program. Hence, a \textit{surface expression} (\textit{surface value}) is an expression (value) that does not contain a \texttt{gated\_ret}.

\textit{Memory isolation.} Now we look at how \( \lambda_{\text{sandbox}} \) uses the privilege mechanism to prohibit the untrusted code from overriding memory of the trusted code. \( \lambda_{\text{sandbox}} \) models coarse-grained memory isolation by splitting the heap into a \textit{high heap} of the trusted code and a \textit{low heap} of the untrusted code via a privilege bit \( p \) in each location. A location that points to the high heap (\textit{i.e.}, whose

\(^4\)All rules of the operational semantics can be found in Appendix A.
privilege bit is High) is called a high location, and similarly a location with \( p = \text{Low} \) is called a low location. We chose to model coarse-grained memory isolation since it is used in most real-world sandboxing techniques [Litton et al. 2016; Vahldiek-Oberwagner et al. 2019; Lamowski et al. 2017; Google 2019]. The parameter \( p \) in alloc\(_p\) specifies in which heap the allocation lands.

O-write enforces the memory isolation by requiring that the current privilege \( p \) is no less than the privilege of the target location \( p' \), i.e., \( p' \subseteq p \). Hence, code running at Low privilege that tries to write to the high heap gets stuck. The operational semantics of alloc, free and CAS prevents the code running at Low from modifying the high heap in a similar way.\(^5\) However, O-read allows code running at Low to read the high heap. This means that while \( \lambda_{\text{sandbox}} \)'s memory isolation enforces integrity of the high heap, it does not enforce confidentiality, as indeed the latter is not always enforced by real-world isolation techniques. Moreover, our goal is to prove robust safety, for which integrity is sufficient.

**System calls and system call assumptions.** Programs running on commodity operating systems interact with their environment — like other programs, the file system, or the internet — via system calls. Thus, from the outside the system calls of a program are its only relevant externally visible behavior. For this reason, \( \lambda_{\text{sandbox}} \) contains a syscall instruction, which is used to define well-behaved programs.

As O-syscall shows, system calls extend the trace \( \pi : \text{list Observation} \) with the observation \( o : (\text{Name}, \text{Lit}, \text{Lit}) \) consisting of the name of the system call \( s \), the argument \( q_a \), and the value returned by the system call \( q_r \). The set of system calls and their behavior are not fixed, but instead they can change depending on the application that is verified. This is implemented by making the stepping relation — and later also the type system — parametric in the system call assumptions \( A : \text{Trace} \rightarrow \text{Prop} \), which encodes whatever assumptions the application makes about how system calls behave. It is used to specify possible traces of system calls. This is not to be confused with the system call policy, which specifies good traces. For example, typing of the file-related systems calls in Figure 4 relies on the open system call returning fresh file handles, which we formalize using the system call assumptions \( A_{\text{file}} \).\(^\text{6}\)

\[
A_{\text{file}} \triangleq \lambda \pi. \forall i_1, i_2, q_{a_1}, q_{a_2}, q_{r_1}, q_{r_2}.
\pi_{i_1} = (\text{open}, q_{a_1}, q_{r_1}) \land \pi_{i_2} = (\text{open}, q_{a_2}, q_{r_2}) \land i_1 \neq i_2 \rightarrow q_{r_1} \neq q_{r_2}
\]

As motivated in Section 2, system call isolation is necessary to prevent the untrusted code from directly issuing malicious system calls. This is enforced in O-syscall by checking that the current privilege \( p \) is High. Prohibiting all system calls is very strict, and some sandboxes used in practice allow the untrusted code to issue some system calls [Vahldiek-Oberwagner et al. 2019; Google 2019]. This can be modeled in \( \lambda_{\text{sandbox}} \) by adding functions to the trusted interface that directly execute a chosen subset of system calls.

**System call policies.** As previously motivated, system calls are the externally visible behavior of a program. But how do we decide if the behavior is good or bad? In this paper, we characterize behaviors, i.e., traces of system calls, as “good” if they conform to some system call policy, which is a predicate on the trace. In contrast to the work by Swasey et al. [2017] we do not use a fixed policy; rather, our program logic and type system are generic with respect to the system call policy, which we specify using an automaton [Schneider 2000; Basin et al. 2013].

**Definition 1.** A system call policy \( SP \) is an automaton consisting of the (possibly infinite) state space \( S \), an initial state \( s_{\text{init}} \in S \), and the transition function \( s \rightarrow s' \), which states that, upon

\(^5\)See Appendix A.

\(^6\)Actually, \( A_{\text{file}} \) also needs to ensure that open only returns integers, but we omit this here.
seeing the observation \( o \), the automaton in state \( s \in S \) steps to the state \( s' \in S' \) (i.e., \( S \) extended with a “bad” state \( \bot \notin S \)). The automaton cannot leave the state \( \bot \), and any state except the bad state is accepting.

For example, the system call policy governing the unsafe_print system call of Section 2 can be seen in Figure 9, and we have already seen the automaton for file-related system calls in Figure 3. The notation \( s(q_a) \leadsto q_r \) denotes the observation \( (s, q_a, q_r) \). All transitions not depicted in the diagram implicitly go to the non-accepting state, which is also implicit.

We write \( s \to \tau \) to assert that \( s \to ? \) and \( \bot \) is not. This is used in the statement of the soundness theorem of the type system in Section 5, which shows that all traces \( \pi \) of a program \( e \) fulfill the system call policy \( SP \) if \( e \) is well-typed with regard to \( SP \).

4 TYPE SYSTEM

Next we show how to verify that a \( \lambda \) sandbox application composed of both trusted and untrusted code is safe. The untrusted code is assumed to be sandboxed using the semantics described in the previous section, but the trusted code is not and needs to be verified for robust safety. As explained in Section 2, our notion of safety is not program progress (non-stuckness), but an application-specific safety property of system call traces (called the system call policy or \( SP \)).

To verify the trusted code, we design a semantic type system inspired by the one of \( \lambda \) Rust [Jung et al. 2018a]. As in RustBelt, our typing judgments are defined semantically using a model built on the Iris separation logic framework in Coq [Jung et al. 2018b]. (Also as in \( \lambda \) Rust, we employ an affine type system, but we omit Rust-specific features like lifetimes and borrowing.) New types can be added by defining their semantics in Iris, and all typing rules are then proven sound individually as lemmas about the typing judgment. This makes it possible to easily extend our type system with new types and typing rules over time. In this section, we describe the overall design of the type system, postponing the details of the model of types to Section 7.

The type system’s basic judgment, \( \nu \triangleleft SP, A \tau \), states that the surface value \( \nu \) has the type \( \tau \). The judgment is parameterized by a system call policy \( SP \) and system call assumptions \( A \). As explained in the previous section, \( A \) represents invariants that govern all system call traces, while \( SP \) is an application-specific safety policy. In the following we sometimes omit \( SP \) and \( A \) to reduce clutter.

Because we use a shallow embedding of typing judgments in our Coq formalization, typing contexts do not bind variables. Rather they collect assumptions about meta-variables \( \nu \), i.e., Coq variables of type \( Val \), which are implicitly universally quantified at the level of the typing rule. The syntax of typing contexts is:

\[
T^{SP, A} ::= \emptyset \mid T^{SP, A}, \nu \triangleleft SP, A \tau
\]

The typing judgment for expressions

\[
T_1 \vdash^{SP, A}_p e : \tau \mid T_2
\]

states informally that, assuming the typing context \( T_1 \), the expression \( e \) is safe (for policy \( SP \)) at privilege \( p \) and will also end its execution at privilege \( p \). Further, if the execution results in a value \( \nu \), then the context for the continuation of \( e \) is context \( \nu \triangleleft SP, A \tau, T_2 \).
The High-Level Benefits of Low-Level Sandboxing

[Image 10x15 to 120x29]

There are two salient points about this judgment. First, it does not stipulate that \( e \) will make progress. It merely ensures safety with respect to SP, which is our only goal. Second, the judgment specifies an output context \( T_2 \), since our type system is affine and thus the execution of \( e \) may result in updates to the typing context.

Despite the type system’s general affineness, values of some types like integers can be copied freely. We call these types copyable and write \( \tau \) copy to mean that \( \tau \) is copyable. This notion is lifted to contexts: \( T \) copy means that all types in \( T \) are copyable.

Figure 10 shows structural typing rules. \( T \)-copy allows duplication of copyable types. \( T \)-val embeds the typing judgments for values into that for expressions. \( T \)-stuck says that a stuck expression \( e \) can be typed at any type. The rule is sound because a stuck expression cannot violate the system call policy and will never evaluate to a value. In the rule \( T \)-let, the \( \forall \) in the second premise is meta-level quantification. It means that the judgment after it is parametric in \( \nu \).

![Fig. 10. Structural typing rules.](image)

![Fig. 11. Selected typing rules for the trusted code.](image)
Next, we turn to rules for specific types, shown in Figure 11. The most basic type is the universal type any, which is inhabited by all values.\footnote{Technically, any is inhabited only by all surface values, but this is implicit since $v \cdot \tau$ is only defined for surface values $v$.} As explained in Section 2, this type is crucial for interaction with untrusted code. The copyable type int is the usual type of integers. (Ground integers are denoted 2.) The rule T-ADD exploits the fact that our type system is not designed to guarantee progress, so the arguments to + can be of type any.

$\lambda_{\text{sandbox}}$ models a low-level language so it does not have native products. Instead products are emulated using the heap. The product type $\Pi(\tau)$ is a pointer to a block of $|\tau|$ values, where the value at position $n$ has the type $\tau_n$. T-ALLOC and T-FREE in Figure 11 show how to allocate and deallocate products. Writing to a product at offset $n$ updates the corresponding type $\tau_n$ to the type of the written value $\tau_w$, as T-WRITE shows. There are two rules for reading from a product. If the read type $\tau_n$ is copyable, it is possible to use T-READ-COPY, which keeps the product unchanged as one would expect. However, if $\tau_n$ is not copyable, $\tau_n$ in the product must be replaced by any to avoid duplication of a non-duplicable type, as T-READ shows. Products themselves are not copyable because they own the memory that stores the elements and this ownership is non-duplicable. Observe how T-FREE consumes the given argument $v$, reflecting this affineness.

Next, we come to the function type, written $\text{fn } \tau_a \rightarrow \tau_r$. Rule T-FN says that a value is a well-typed function of type $\text{fn } \tau_a \rightarrow \tau_r$ if the body of the function $f$ substituted with values $\tau_a$ of the argument types $\tau_a$ is a well-typed expression of the return type $\tau_r$. The body’s typing is also allowed to assume that all recursive occurrences $f$ have the correct type. We insist that function types be copyable as this allows functions to be called multiple times, which is convenient in practice. This means that the typing context $T$ used to type the function in T-FN must be copyable.

**System calls.** An interesting aspect of typing is system calls, since this is where the policy SP and the system call assumptions $A$ become relevant. Clearly, it is impossible to write meaningful rules for system calls independent of SP and $A$, but given a pair of SP and $A$, specific rules for system calls can be proved sound. As an example, Figure 12 shows typing rules for the system calls introduced in Section 2 using the LessThan1000 and file types. Here, printSP and fileSP are the two safety policies described in Section 2, and $A_{\text{file}}$ is the system call assumptions for the file-related system calls defined in Section 3, which ensures that open always returns a fresh identifier.

The rule for unsafe\_print is self-explanatory. The rules for open, read, and close make use of the fact that the concrete representation of the abstract type file in our model coincides with the type of file handles returned by open. Hence, we can type these system calls using the type file. The rules also prevent the use of a closed file instance since close consumes its argument. Under the hood, the soundness of these rules relies on Iris-level ghost state. For instance, we use this ghost state to track the set of file handles that are currently open. The interpretation of file owns one of the file handles in this set, and close removes the file handle from this set, so we know in the precondition of read and close that the passed handle could not have been closed previously. file is not copyable since the ownership of the file handle cannot be duplicated. We defer the formal details of this proof to Section 7.

**Interacting with untrusted code.** A second interesting aspect of our type system is the rule for interacting with untrusted code. To call untrusted code, trusted code uses the gated\_call instruction with the following typing rule:

$$T\text{-gated-call } \overline{v} \cdot \text{any} \vdash_{\text{High}} \text{gated-call}(s, \overline{v}) : \text{any} \mid \emptyset$$

The rule says that it is safe (for any safety property SP) to call arbitrary code, as long as the called code is sandboxed (forced by gated\_call) and nothing is assumed about the result (the result is
of type \( \text{any} \). Intuitively, the rule is sound because sandboxing prevents the untrusted code from directly making any system calls (system call isolation) or modifying any private memory of the trusted code (memory isolation). We expand this intuition formally in Section 5.

Our Coq development \cite{Samler19} contains many other types like sums, maps, and mutexes, the details of which we omit here due to lack of space.

5 ROBUST SAFETY OF THE TYPE SYSTEM

Robust safety of our type system has two parts. First, we justify the typing rule \( T\text{-gated-call} \) by proving in Theorem 2 that any surface expression is semantically well-typed at \( \text{Low} \) privilege. This allows well-typed programs to make gated calls to untrusted code. Second, Theorem 3, the soundness theorem of our type system, shows that a program that is well-typed w.r.t. a system call policy \( SP \) will uphold that policy no matter what code runs inside the sandbox. Together, these theorems entail that well-typed code is robustly safe, i.e., is guaranteed to uphold the system call policy even when interacting with arbitrary untrusted code via \( \text{gated_call} \).

To show that arbitrary code is safe to run inside the sandbox, we prove that all surface expressions \( e \) are typeable at \( \text{Low} \). We do not know anything about the value that \( e \) will evaluate to, so we can only assign the type \( \text{any} \):

\[
\emptyset \vdash_{\text{Low}} e : \text{any} \mid \emptyset
\]

The proof of Theorem 2 requires several assumptions about the model of the type system:

1. The trusted code may only have invariants on the high heap, not on the low heap. We need this restriction since memory isolation only protects the high heap, and the untrusted code can manipulate the low heap in arbitrary ways.

2. The functions in the interface of the trusted code are safe to call at any time with any value, i.e., are of type \( \text{fn} \text{any} \rightarrow \text{any} \).

With these assumptions it is possible to show Theorem 2 by induction over \( e \) since the memory isolation protects the invariants of the trusted code on the high heap, system call isolation prevents the untrusted code from issuing malicious system calls, and assumption (2) ensures that each \( \text{gated_call} \) of the untrusted code can be handled safely.
Soundness of T-gated-call follows directly from Theorem 2. After executing the gated_call with O-gated-call, the privilege is Low and given a function retrieved from the interface $I$, one must show that its application to the supplied arguments $\overline{s}$ is safe to execute. However, this is exactly what Theorem 2 shows. Its safe execution results in a value of type any, which is returned to the trusted code after the privilege switches back to High due to the gated_ret.

Robust safety. Now we can state the robust safety theorem, i.e., the soundness theorem for our type system, which is a variant what is typically called “adequacy” in semantic soundness proofs. It shows that the trace of all executions of a well-typed program satisfies the system call policy for arbitrary untrusted code inside the sandbox:

**Theorem 3 (Robust Safety).** For

- any untrusted code represented as arbitrary surface values $\overline{v}_u$ with names $\overline{s}_u$,
- any exposed functions $\overline{v}_t$ of the trusted code with names $\overline{s}_t$,
- any system call policy $SP$ with initial state $s_{SP}^{init}$,
- any system call assumptions $A$,
- any map from block names to initial values of global variables $m : Name \rightarrow list Lit$,
- any value $main$, and
- any initial context $T_{init}$,

the following rule is sound:

$$
(3) T_{init} \vdash_{SP,A}^{High} main : fn \ [\ ] \rightarrow any \ | \ \emptyset \\
(4) \forall u \in \overline{v}_t. T_{init} \vdash_{SP,A}^{High} u : fn any \rightarrow any \ | \ \emptyset \\
(5) I = [(Low, s_u) \mapsto \overline{v}_u, (High, s_t) \mapsto \overline{v}_t] \\
(6) I = h_{init} m, [\ ] | main () at High \rightarrow_\Lambda h, \pi | E \\
(7) \exists st. st_{SP}^{init} \xrightarrow{\pi} st_{SP}
$$

The main goal of this theorem is to show that every trace $\pi$ produced by executing $main$ (assumption (6)) fulfills the system call policy $SP$ (conclusion (7)), which is the definition of good execution used in this paper. The rest of the theorem is a mouthful, so let us walk through it.

Assumption (5) tells us that the trusted code exposes the functions $\overline{v}_t$, while the untrusted code consists of the surface values $\overline{v}_u$. When using this theorem, as we are going to do in Section 6, one provides concrete values for $\overline{v}_t$ while keeping $\overline{v}_u$ universally quantified. This means that $main$ is robustly safe against linking with arbitrary untrusted code $\overline{v}_u$. Assumptions (3) and (4) state that the trusted code must be well-typed and thus verified; the latter is necessary to fulfill assumption (2) of Theorem 2.

Assumptions (1), (2), $T_{init}$, and $m$ are used to implement global variables. Some wrappers (e.g., dynamic sealing) need global variables to retrieve trusted state (e.g., the global sealing table), as seen in Section 2. These global variables are represented as hard-coded locations in the trusted code via the function $g2l : Name \rightarrow Loc$, which turns the name of the global variable into the corresponding location. The map $m$ specifies the initial values for the global variables, which are used to create the initial heap $h_{init} m$. The global variables can be used in (3) and (4) via the initial context $T_{init}$. $InitFromHeap(m, T_{init})$ in (2) informally states that $T_{init}$ can be established from the initial heap $h_{init} m$. Assumption (1) is necessary in order to copy $T_{init}$ so that it can be used in both (3) and (4), and T-FN in Figure 11 requires $T_{init}$ to be copyable anyways.

Given our careful technical setup, Theorem 3 is not actually difficult to prove since the semantic model of typing (see Section 7) is defined to make it fall out naturally. In particular, typability of $main$ implies it is safe to execute and thus upholds the system call policy.
6 WRAPPERS

The previous section showed that the whole program is safe if all interaction with untrusted code happens at the \texttt{any} type. However, the \texttt{any} type is difficult to work with because it does not provide any useful information. In order to more effectively integrate the sandbox into a typed application, we want to make it possible for trusted code to interact with untrusted code using more richly typed interfaces, but without sacrificing robust safety.

To this end, we propose a system of automatic checks and transformations that allows a programmer to write a trusted function with an informative type $\texttt{fn } \tau \rightarrow \tau_r$, and automatically lift it to a function of the type $\texttt{fn any } \rightarrow \texttt{any}$, which can be shared with untrusted code. Dually, it allows an untrusted function of type $\texttt{fn any } \rightarrow \texttt{any}$ to be lifted automatically to the type $\texttt{fn } \tau \rightarrow \tau_r$, which can then be applied safely at that type by trusted code.

This lifting mechanism works through a system of \texttt{wrappers} converting values to (or from) the \texttt{any} type. We provide several examples of wrappers as a library, but the programmer can define and verify custom wrappers for whatever new types they define, thus making those types available for automatic transformation between trusted and untrusted code (this is a one-time effort for every new type). The verification of a wrapper works by proving that it semantically inhabits a specific function type.

More concretely, one can associate one or both of the following functions with each type $\tau$:

- $\tau\texttt{.import} < \texttt{fn any } \rightarrow \tau$: A type $\tau$ can provide an import function, which is used to check and transform values that the untrusted code passes to the trusted code. Such a type is then called \textit{importable}. Simple types like \texttt{LessThan1000} from Section 2 can be imported by checking if the untrusted code passed a value of the type $\tau$ and either passing the value to the trusted code or getting stuck.

- $\tau\texttt{.export} < \texttt{fn} \tau \rightarrow \texttt{any}$: By providing an export function a type $\tau$ can state how it should be transformed when values of this type are passed from the trusted to the untrusted code. Such a type is then called \textit{exportable}. It is always possible to implement $\tau\texttt{.export}$ as the identity function since \texttt{any} is inhabited by all values. However, we will see that export is still useful, e.g., for transforming values to a representation which works inside the sandbox or converting them to a form that can be checked by import.

If a function uses import or export of a type parameter, this function implicitly is only defined if the type parameter is importable or exportable, respectively. The Coq formalization implements this mechanism by providing \texttt{Import} and \texttt{Export} type classes that can be implemented for importable and exportable types. For example, an implementation in Rust could use traits to associate a function with a type.

Note that import and export can be arbitrary functions, so long as they are well-typed. This is unlike the setting of higher-order contracts [Findler and Felleisen 2002; Blume and McAllester 2004] and gradual typing where, usually, wrappers can only check values but cannot modify them, and must maintain some additional properties, e.g., that importing an exported value should return the original value [New and Ahmed 2018]. In our setting such restrictions would rule out useful wrappers like dynamic sealing, as we will see later.

Figure 13 shows how the wrappers are used. The \texttt{wrap_import} macro can be used by the trusted code when it wants to call the function $s$ exposed by the untrusted code. It first calls the corresponding $\tau_s\texttt{.export}$ function on each argument to transform all values in $x$ to \texttt{any}. Then \texttt{wrap_import} calls the untrusted code using \texttt{gated_call}. This call is safe due to \texttt{T-gated-call} and returns a value of type \texttt{any}, which is checked and transformed to a value of type $\tau_r$ using $\tau_r\texttt{.import}$. In total, this means that \texttt{wrap_import} can be typed at type $\texttt{fn } \tau \rightarrow \tau_r$, as Figure 13 shows.
\begin{align*}
\text{wrap}_\text{import} & \triangleq \lambda x. \text{let } x = \tau_a.\text{export}(x); \text{let } r = \text{gated}_\text{call}(s, x); \tau_r.\text{import}(r) \\
\text{wrap}_\text{export} & \triangleq \lambda x. \text{let } x = \tau_a.\text{import}(x); \text{let } r = v(x); \tau_r.\text{export}(r)
\end{align*}

\begin{tabular}{l}
\text{T-wrap-import} \\
$T_r \models_{\text{High}} \tau_r.\text{import}: \tau_{\text{fn}} \rightarrow \tau_a | \emptyset$	\
$T_a \models_{\text{High}} \tau_a.\text{export}: \tau_a \rightarrow \tau_r | \emptyset$	\

\hline
$T_a, T_r \models_{\text{High}} \text{wrap}_\text{import} s \tau_a \tau_r : \tau_{\text{fn}} \rightarrow \tau_r | \emptyset$
\end{tabular}

\begin{tabular}{l}
\text{T-wrap-export} \\
$T_a \models_{\text{High}} \tau_a.\text{import}: \tau_{\text{fn}} \rightarrow \tau_a | \emptyset$	\
$T_r \models_{\text{High}} \tau_r.\text{export}: \tau_r \rightarrow \tau_{\text{fn}} | \emptyset$	\

\hline
$T_a, T_r \models_{\text{High}} \text{wrap}_\text{export} v \tau_a \tau_r : \tau_{\text{fn}} \rightarrow \tau_{\text{fn}} | \emptyset$
\end{tabular}

\text{Fig. 13. Applying wrappers to functions. The overline notation is explained in the main text.}

\begin{align*}
\text{any.} \text{import} & \triangleq \lambda x. x \leftarrow \text{fn any} \rightarrow \text{any} \\
\text{int.} \text{import} & \triangleq \lambda x. x + 0 \leftarrow \text{fn any} \rightarrow \text{int} \\
\text{any.} \text{export} & \triangleq \lambda x. x \leftarrow \text{fn any} \rightarrow \text{any} \\
\text{int.} \text{export} & \triangleq \lambda x. x \leftarrow \text{fn int} \rightarrow \text{any}
\end{align*}

\text{Fig. 14. import and export wrappers for any and int.}

\begin{tabular}{l}
\text{T-low-copy} \\
$\text{LowPtr copy}$	\
$\emptyset \vdash_{p} \text{alloc}_\text{Low}(z) : \text{LowPtr} | \emptyset$
\end{tabular}

\begin{tabular}{l}
\text{T-free-low} \\
$v \leftarrow \text{LowPtr} \vdash_{p} \text{free}(z, v) : \text{any} | \emptyset$
\end{tabular}

\begin{tabular}{l}
\text{T-write-low} \\
$v_w \leftarrow \text{any}, v_p \leftarrow \text{LowPtr} \vdash_{p} v_p, z \leftarrow v_w : \text{any} | \emptyset$
\end{tabular}

\begin{tabular}{l}
\text{T-read-low} \\
$v \leftarrow \text{LowPtr} \vdash_{p} \text{lv}, z : \text{any} | \emptyset$
\end{tabular}

\text{Fig. 15. Typing rules for LowPtr}

The trusted code can use \text{wrap}_\text{export} when it wants to expose a function \(v\) to the untrusted code. The sequence of arguments \(x\) is checked and transformed to values of the correct type using \(\tau_a.\text{import}\). One can thus call \(v\), which expects arguments of types \(\tau_a\). After the function call the result \(r\) is exported to the untrusted code using \(\tau_r.\text{export}\). All in all, the signature of a function wrapped by \text{wrap}_\text{export} is \(\text{fn any} \rightarrow \text{any}\), which can be safely exposed to the untrusted code.

We define \text{wrap}_\text{import} and \text{wrap}_\text{export} using overline notation to support an arbitrary number of arguments. For example, the over-lined \texttt{let} \(x = \tau_a.\text{export}(x)\) macro expands to

\begin{verbatim}
\texttt{let } x_1 = \tau_a.1.\text{export}(x_1); \texttt{let } x_2 = \tau_a.2.\text{export}(x_2); \ldots
\end{verbatim}

\textit{Examples of wrappers.} Some of the simplest wrappers are the wrappers for \texttt{any} and \texttt{int}, which can be seen in Figure 14. The wrappers for \texttt{any} are trivial since both are just the identity function.

The wrapper for importing an integer adds 0 to the value passed by the untrusted code. This ensures that it is an integer since addition gets stuck otherwise. Thus \(\text{int.}\text{import} \triangleq \text{fn any} \rightarrow \text{int}\) by \text{T-add}. Exporting an \texttt{int} simply returns it unchanged to the untrusted code.
LowPtr. Importing locations is more difficult than importing integers because the untrusted code can confuse the trusted code by passing a high location where the trusted code expects a low location. The memory isolation does not prevent the trusted code from writing to the location since it is running at High privilege.

To prevent this confused deputy attack [Hardy 1988] we use our extensible type system to define a new type LowPtr of low locations:

\[
T_{\text{LowPtr}} = (\text{Low}, \_\_\_), \quad l \triangleq \text{LowPtr}
\]

The typing rules for LowPtr are shown in Figure 15. Since there are no invariants on the low heap, it is safe to read from and write to a LowPtr using T-read-low and T-write-low. However, the trusted code cannot expect that a LowPtr points to any specific value even after reading it since the untrusted code can concurrently change the value on the low heap. Thus T-read-low returns any. This means that working with LowPtr requires some care, and it is mainly useful as a building block for implementing import and export for other types, as we will see when introducing the wrappers for \(\Pi(\tau)\).

Crucially, LowPtr is importable as Figure 16 shows. LowPtr.import uses get_priv to check that the untrusted code actually passed in a low location and gets stuck otherwise.

There are two subtle points about LowPtr that we wish to highlight here. First, LowPtr.import relies on the get_priv instruction; thus, a concrete sandboxing technique that wants to use the results of this work must provide a way to distinguish a low location from a high one. Second, LowPtr relies on the fact that a low location will never become high, even if it is deallocated and later reallocated. Otherwise there would be a possibility of a time-of-check to time-of-use attack [Abbott et al. 1976] between LowPtr.import and a subsequent write to the LowPtr. A similar point was made by de Amorim et al. [2018] regarding pointer reuse in memory-safe languages.

Both requirements are trivially fulfilled when using an isolation technique based on multiple address spaces, since each pointer is permanently associated with one address space. However, programmers usually also respect these requirements even with single address-space isolation techniques as it simplifies informal reasoning about the code [Vahldiek-Oberwagner et al. 2019]. For example, the Linux kernel has a fixed partitioning of the address space into a user range and a kernel range and uses this to check pointers passed from the untrusted (user-space) code.

Products. The wrappers for the \(\Pi(\tau)\) type can be seen in Figure 17. Its import reads \(\|\tau\|\) values from the low heap, imports them with the imports of the corresponding types, and writes the imported values to a product on the high heap. Its export works the other way around by writing the exported values of the product to the low heap. The original product is freed because the caller gave ownership of the product to the wrapper, and thus also the responsibility for freeing its allocation.

Notice that \(\Pi(\tau).\text{import}\) (and \(\Pi(\tau).\text{export}\)) is a macro similar to wrap_import and wrap_export to support products of arbitrary size. The expression \(l.i \leftarrow \tau.\text{import}(\!u.\!i)\) is repeated for all types \(\tau\) in \(\pi\) with \(i\) referring to the index of \(\tau\) in \(\pi\).

The most obvious difference from the previous wrappers is that the \(\Pi(\tau).\text{export}\) function is not just the identity. Products are implemented as allocations on the high heap. Simply passing a high
location to the untrusted code is not very useful since the untrusted code could neither write to it nor free it. Hence, the export function copies the exported values to the low heap so that the untrusted code can manipulate them freely. This copying, which is also done by default by existing tools [Google 2019; Lamowski et al. 2017], can be expensive, and depending on the use-case more optimized patterns like dynamic sealing are possible.

The import function follows a similar pattern to export, but here it is vital for safety to copy the values from the low to the high heap, because otherwise the values could be changed after they were checked. To check the components of the product, \( \Pi(\tau) \).import uses the import function of \( \tau \in \tau \). This generic approach to wrapping means that the wrappers compose nicely and \( \Pi(\tau) \) is importable (exportable) if all types in \( \tau \) are importable (exportable).

\textit{unsafe_print example.} Now we have everything in place to revisit the unsafe_print example of Section 2 and show that main\_file function is safe to execute for arbitrary untrusted code \( v_u \) inside the sandbox when exposing the wrapped typed_print function. Applying Theorem 3 with an empty set of global variables, an empty initial context, and no system call assumptions (i.e., \( A \triangleq \lambda \pi. \text{True} \)) results in the obligations

\[
\begin{align*}
  & \text{main\_sandbox} \triangleright_{\text{printSP}} A \text{ fn } [\_] \rightarrow \text{any} \quad \text{typed\_print} \triangleright_{\text{printSP}} A \text{ fn } \text{LessThan1000} \rightarrow \text{any} \\
  \text{which are easy to show with the typing rules described so far. All in all, we get}
\end{align*}
\]

\[
I = [(\text{Low}, s_u) \mapsto v_u, (\text{High}, \text{typed\_print}) \mapsto \text{wrap\_export } \text{[LessThan1000]} \text{ any } \text{typed\_print}] \\
I \vdash \emptyset, [\_] \mid \text{main\_sandbox} () \text{ at } \text{High} \rightarrow^* A h, \pi \mid E
\]

\[
\exists st. st^\text{init}_{\text{printSP}} \triangleright_{\text{printSP}} A st^\text{\check{\nu}}
\]

This says that if we execute main\_sandbox (), which invokes the untrusted\_main function of the arbitrary untrusted code \( v_u^8 \), which in turn may invoke the \text{wrapped} version of typed\_print, then the trace \( \pi \) of that execution adheres to the system call policy printSP.

\textit{Dynamic sealing.} As motivated in Section 2 with the file example, there are types where simple inspection cannot determine whether a value belongs to the type, e.g., integers representing valid file handles. A popular solution to enforce the invariants of objects that are passed to untrusted

\footnote{If the untrusted code does not expose a function named untrusted\_main, this call gets stuck, which is not a problem.}
The important point about Sealed$_{s,\tau}$ is that it is importable and exportable for all types $\tau$, even if $\tau$ is not importable or exportable itself. It can be seen as a marker type for wrap_import and wrap_export to specify that some arguments or return values should be wrapped via dynamic sealing instead of their own wrappers (if they have any). T-sealed makes sure that Sealed$_{s,\tau}$ can be treated as a value of type $\tau$ during verification and thus Sealed$_{s,\tau}$ does not influence the verification of the trusted code.

Figure 18 shows how Sealed$_{s,\tau}$ implements import and export using a global map that is stored as a global variable in block $s$. The function export puts the exported value in the map and gets a
fresh seal in return. This seal is then returned to the untrusted code. The function import interprets its argument as a seal and looks it up in the map. If this lookup succeeds, import takes the sealed value out of the map and returns it to the trusted code. If the untrusted code passed an invalid seal (a seal that is not in the global map), import gets stuck. Thus, the untrusted code can only pass back seals for values that have been handed out by the trusted code earlier. Since import and export might be called concurrently, a mutex is necessary to guard access to the map.

Figure 19 shows the additional types and functions used by the wrappers for Sealed$_{s,τ}$.

- **mutex$_{τ}$** is a simple spin lock based mutex, which guards a value of type $τ$. The value can be taken out of the mutex with acquire, which spins until it can lock the mutex. In addition to the value of type $τ$, acquire also returns a guard$_{τ}$, which release puts back a value of type $τ$ into the mutex. The logic of the mutex ensures that only one thread at a time can access the guarded value and thus mutex$_{τ}$ is copyable independent of $τ$ copy.

- **map$_{τ}$** is a map from integer keys to values of type $τ$. The function map_put stores a value of type $τ$ in the map and returns the updated map and the index of the newly inserted value. The function map_take can then be used to take the value at the specified key out of the map. It returns option$_{τ}$ because the key might not be in the map. The function unwrap returns the value contained in the option if there is any, and gets stuck otherwise.

The implementations and proofs for these types and functions can be found in the Coq development [Sammler et al. 2019]. Let us revisit the file system example of Section 2 to see how these wrappers for dynamic sealing are used in action. The goal is to show that it is safe to expose the wrapped do_open, do_read, and do_close functions to arbitrary untrusted code inside the sandbox.

Sealed$_{s,τ}$ uses a global variable to store the map used for sealing. We pick the block denoted by the name file to store this map and fill this block in the initial heap with an empty map $q_{\text{init}}$:

$$m \triangleq \{ \text{file} \mapsto q_{\text{init}} \}$$

This means that we have to show

$$I = \begin{align*}
(\text{Low}, s_I) & \mapsto v_u, \\
(\text{High}, \text{do_open}) & \mapsto \text{wrap_export} \ [\text{int}] \text{Sealed_{file, file}} \text{do_open}, \\
(\text{High}, \text{do_read}) & \mapsto \text{wrap_export} \ [\text{Sealed_{file, file}} \Pi (\text{Sealed_{file, file}}, \text{any}) \text{do_read}, \\
(\text{High}, \text{do_close}) & \mapsto \text{wrap_export} \ [\text{Sealed_{file, file}} \text{any} \text{do_close}
\end{align*}$$

$$I + h_{\text{init}} m_s \ [\ ] \ | \ \text{main}_{\text{file}} () \mid \text{at High} \cdots \sigma^*_{\text{file}} h', \pi \mid E$$

where the main function main$_{\text{file}}$ directly calls the untrusted code

$$\text{main}_{\text{file}} \triangleq \lambda \_ . \ \text{gated\_call}(\text{untrusted\_main}, ())$$

After applying Theorem 3 and establishing the initial context

$$T_{\text{init}} \triangleq g\_2l \ \text{file} \triangleq \text{mutex}_{\text{map}_{\text{file}}}$$

one can use T-wrap-export together with the typing rules for the wrappers to type the exposed functions. $T_{\text{init}}$ is used by T-sealed-import and T-sealed-export. Then it is only necessary to show the following:

$$\begin{align*}
\text{main}_{\text{file}} & \triangleq \text{file}_{\text{SP}} \ \text{fn} \ [ ] \rightarrow \text{any} \\
\text{do\_open} & \triangleq \text{file}_{\text{SP}} \ \text{fn} \ \text{Sealed}_{\text{file, file}} \rightarrow \Pi (\text{Sealed}_{\text{file, file}}, \text{any}) \\
\text{do\_read} & \triangleq \text{file}_{\text{SP}} \ \text{fn} \ \text{Sealed}_{\text{file, file}} \rightarrow \Pi (\text{Sealed}_{\text{file, file}}, \text{any}) \\
\text{do\_close} & \triangleq \text{file}_{\text{SP}} \ \text{fn} \ \text{Sealed}_{\text{file, file}} \rightarrow \text{any}
\end{align*}$$

The type of main$_{\text{file}}$ follows from T-gated-call; the others are easy to show by first replacing Sealed$_{\text{file, file}}$ with file using T-sealed and then using the typing rules in Figure 12.
7 MODEL OF THE TYPE SYSTEM

This section describes the model of the type system that we use to prove the rules from the previous sections sound. The idea of using such a semantic type system and the model itself are heavily inspired by RustBelt [Jung et al. 2018a], which was in turn inspired by prior work on Foundational Proof-Carrying Code [Ahmed et al. 2010]. However, our model is significantly simpler since we do not deal with Rust-specific features like lifetimes or borrowing.

As in Jung et al. [2018a], types are predicates on values in a separation logic based on Iris [Jung et al. 2018b], a framework for building higher-order concurrent separation logics in Coq. We instantiate Iris with \( \lambda_{\text{sandbox}} \) to attain the program logic Iris\(_{\text{sandbox}}\). Like the type system, Iris\(_{\text{sandbox}}\) and thus also the type of its propositions \( iProp_{\text{sandbox}}^{SP,A} \) are parameterized by a system call policy \( SP \) and system call assumptions \( A \).

**Definition 4.** A type \( \tau \) is a predicate \( \llbracket \tau \rrbracket^{SP,A} \in SurfaceVal \rightarrow iProp_{\text{sandbox}}^{SP,A} \). We write \( \nu \not\in^{SP,A} \tau \) for \( \llbracket \tau \rrbracket^{SP,A} \nu \).

Defining types as predicates in Iris makes the type system affine since Iris is an affine separation logic, which means that it enjoys the weakening rule \( P \Rightarrow Q \Rightarrow P \) [Jung et al. 2018b] used to show T-weaken in Figure 10. Iris and thus Iris\(_{\text{sandbox}}\) provide an extensible notion of ghost state, which can be used in the definition of types. For example, ghost state is useful to define the \( \text{file} \) type from Section 2, as we will see later. Here, we will not explain exactly how ghost state is modeled, but rather refer interested readers to the paper of Jung et al. [2018b] for the details.

As usual in an affine setting, some typing rules given in earlier sections involved the notion of copyability. Recall from Section 4 that a type \( \tau \) is copyable if \( \nu \not\in \tau \) is freely duplicable. In the model this is realized by defining a type \( \tau \) as copyable if \( \nu \not\in \tau \) is persistent (in Iris lingo), which means it does not express exclusive ownership of resources and thus can be freely duplicated.

The semantic interpretation of a typing context \( \llbracket T \rrbracket \) is defined as the separating conjunction of all \( \nu \not\in \tau \) in \( T \). The definition of the typing judgment for expressions is based on Iris Hoare triples.

**Definition 5.** The typing judgment for an expression \( e \) is defined as follows:

\[
\llbracket T_1 \rrbracket^{SP,A} e : \tau \mid T_2 \triangleq \begin{cases}
\llbracket T_1 \rrbracket^{SP,A} e \at p \{ \nu \at p, \nu \not\in^{SP,A} \tau \ast \llbracket T_2 \rrbracket^{SP,A} \}\,
\end{cases}
\]

The Hoare triple asserts that \( e \) is safe to run at privilege \( p \) if \( \llbracket T_1 \rrbracket \) holds. If the execution of \( e \) terminates in a value \( \nu \), the privilege will be \( p \), \( \nu \) will have type \( \tau \), and \( \llbracket T_2 \rrbracket \) will hold. The \( ? \) in Definition 5 signifies that \( e \) is allowed to get stuck. Safe to run means that \( e \) upholds all invariants. In particular, upholding the system call policy \( SP \) is an invariant of Iris\(_{\text{sandbox}}\), which means that no well-typed expression can violate \( SP \). This is the main ingredient for the proof of Theorem 3. These definitions are already enough to show the typing rules in Figure 10 sound.

The definitions of \texttt{any} and \texttt{int} are straightforward as they are defined to be inhabited by all surface values and all integers, respectively:

\[
\llbracket \text{any} \rrbracket = \lambda \nu. \text{True} \quad \llbracket \text{int} \rrbracket = \lambda \nu. \exists z : \mathbb{Z}. \nu = z
\]

As motivated in Section 4, products are stored on the high heap. As usual in separation logic, Iris\(_{\text{sandbox}}\) expresses ownership on the heap using the points-to-predicate \( l \mapsto \overline{v} \), which asserts that \( l \) points to the consecutive values \( \overline{v} \). This predicate allows the trusted code to keep invariants on the heap, which are upheld when executing the untrusted code inside the sandbox. However, in Iris\(_{\text{sandbox}}\), \( l \mapsto \overline{v} \) is only defined for high locations \( l \). Thus, it is not possible to have invariants on
the low heap, which is necessary to fulfill assumption (1) of Theorem 2. Using this predicate, Π(τ) is defined as follows:

\[ \Pi(\tau) \triangleq \lambda v. \exists l. v = l \cdot \exists l_p. l \mapsto \overline{\nu}_p \cdot \nu_p \cdot \tau \cdot \text{DeallocSize}(l, |\tau|) \]

The proposition \text{DeallocSize}(l, |\tau|) is the same as in Jung et al. [2018a]; it manages the right to deallocate the location \( l \).

The type of functions \( \text{fn} \rightarrow \tau \) is defined in the same way as in Jung et al. [2018a] and closely follows T-FN in Figure 11.10

\[ \Pi(\tau) \triangleq \lambda v. \exists \overline{x} e. v = (\text{rec } f \overline{x}) = e \cdot \forall \nu_a. \nu_a \cdot \tau_a \vdash \text{High} e[f/\overline{x}, \nu_a/\overline{x}] : \tau_r \mid \emptyset \]

The modality \( \Box \) ensures that only persistent resources are used in the proof of \( v \in \text{fn} \rightarrow \tau \), which is necessary to show T-FN-COPY. The Coq formalization [Sammler et al. 2019] shows how the typing rules in Figure 11 can be proven sound using these definitions.

As T-LOW-PTR already suggests, LowPtr is defined as all low locations:

\[ \text{LowPtr} \triangleq \lambda v. \exists l. v = l \cdot (\text{Low}, \_ , \_ ) \]

The soundness of the typing rules in Figure 15 follows from the fact that low locations cannot be owned and getting stuck is safe.

Typing of system calls. As the concluding part of our tour of the model of the type system, we look at how one can prove typing rules for system calls like T-PRINT and the file-related typing rules in Figure 12. As mentioned in Section 4, there is no generic typing rule for system calls; rather, one proves typing rules for specific system calls by reasoning in the model.

Showing that a system call is safe to execute reduces to a proof that the automaton of the system call policy can make the corresponding step for the supplied arguments. In the case of T-PRINT, for example, this is easy because one knows that \( v \) is a value less than 1000, which is always safe to pass to unsafe_print.

In contrast, safety of an execution of read or close depends on the state of the system call policy. The typing rules T-FILE-READ and T-FILE-CLOSE rely on the file type, which represents an open file. But how to define such a type in the model? The solution to this problem is to extend the specification of each system call policy \( SP \) with an invariant on the state of \( SP \):

\[ I_{SP} : S_{SP} \rightarrow iProp_{sandbox}^{SP,a} \]

Using \( I_{SP} \), it is possible to show the following simplified proof rule for system calls:11

\[ \text{P-syscall} \]

\[ P(s, q_a, \Phi) \triangleq \forall q_r. \text{st}. I_{SP} \text{st} \Rightarrow \exists s'. \text{st}' \cdot (s, q_a, q_r) \cdot p_{SP} \text{st}' \cdot * \Phi \cdot q_r \]

\[ \{ P(s, q_a, \Phi) \} \text{syscall}(s, q_a) \text{at High} \{ q_r \text{ at High} \cdot \Phi \cdot q_r \} \]

This rule states that a system call \( s \) with argument \( q_a \) is safe to execute if, assuming \( I_{SP} \) holds for the current state \( s \) of the system call policy, \( SP \) accepts \( s \) for all return values \( q_r \) by stepping to a new state \( s' \neq \bot \) and \( I_{SP} \) holds for the new state \( s' \).

The state of fileSP is the finite set of open file handles \( st \in \mathcal{P}(Z) \). To define \( I_{fileSP} \) and the file type, we use two new kinds of ghost state called file_auth and file_token \( z_f \), which are governed.

---

10For purposes of presentation, we suppress the \( * \) modality, which is necessary to prove T-FN for recursive functions.

11To a first approximation, readers unfamiliar with Iris can read the view shift \( \Rightarrow \) as if it were the magic wand \( \rightarrow \).
by the following rules:\footnote{The definition of file_auth \( st \) and file_token \( z_f \) and the proofs of their rules can be found in the Coq development \cite{Sammler2019}.

\begin{align*}
& \textbf{P-file-open} \quad z_f \notin \text{st} \\
& \text{file_auth} \; st \implies \text{file_auth} \; (st \cup \{z_f\}) * \text{file_token} \; z_f \\
& \textbf{P-file-in} \quad \text{file_auth} \; st * \text{file_token} \; z_f \iff z_f \in st \\
& \textbf{P-file-close} \quad \text{file_auth} \; st * \text{file_token} \; z_f \implies \text{file_auth} \; (st \setminus \{z_f\})
\end{align*}

Conceptually, \text{file_auth} \; st asserts that the set of open files is currently \( st \) and \text{file_token} \; z_f states that \( z_f \) is in this set (as \text{P-file-in} shows) and thus represents an open file. To close a file, which means removing it from \( st \), one has to provide the corresponding \text{file_token} \; z_f, as can be seen in \text{P-file-close}. \text{I}_{\text{fileSP}} \) maintains \text{file_auth} \; st, and \text{file} is defined using \text{file_token} \; z_f:

\[
\text{I}_{\text{fileSP}} \triangleq \lambda \text{st. file_auth} \; \text{st} \quad \text{[file]} \triangleq \lambda \text{v. } \exists z_f. \; \text{v} = z_f * \text{file_token} \; z_f
\]

With these definitions we have everything in place to prove \text{T-file-read} and \text{T-file-close}. For \text{T-file-read}, after unfolding the definition of the typing judgment and applying \text{P-syscall}, one combines \text{file_token} \; z_f \) from \text{v * file} \) with \text{file_auth} \; st \) from \text{I}_{\text{fileSP}} \) to learn that the argument \( z_f \) to read is in the open state and thus \text{fileSP} \) can make a step. The proof of \text{T-file-close} \) is almost the same except that close removes \( z_f \) from the set of open files and thus one must use \text{P-file-close} \) to update \text{file_auth} \; st \) to \text{file_auth} \; (st \setminus \{z_f\}). This consumes \text{file_token} \; z_f \), which is the reason why \text{T-file-close} \) consumes \text{v * file}.

Unfortunately, the simplified version of \text{P-syscall} stated above is not quite strong enough to prove \text{T-file-open}. To use \text{P-file-open} \) we must show that \( z_f \) is not already an open file. Otherwise we could use \text{P-file-open} \) to generate arbitrarily many \text{file_token} \; z_f \) and use them to close the same file handle multiple times. Thus, we need to assume that the open system call returns fresh file handles using the system call assumptions \( A_{\text{file}} \) defined in Section 3. However, as stated above, \text{P-syscall} \) does not make use of \( A_{\text{file}} \), and thus to prove \text{T-file-open} \) we will need a stronger version of \text{P-syscall}.

Before giving the stronger \text{P-syscall-full} rule, we first need to introduce two new kinds of ghost state for reasoning about the trace:

- \text{full_trace} \( \pi \) asserts that the current trace of the execution so far is \( \pi \). It also contains the permission to extend the trace and thus cannot be duplicated.
- \text{in_trace} \( i \; o \) states that the observation at index \( i \) of the current trace is \( o \). Since the trace is append-only, \text{in_trace} \( i \; o \) is a persistent assertion, which can be freely duplicated.

The important property of \text{full_trace} \( \pi \) and \text{in_trace} \( i \; o \) is that they can be combined to learn \( \pi_i = o \).

\begin{align*}
& \textbf{P-in-trace} \quad \text{full_trace} \; \pi * \text{in_trace} \; i \; o \iff \pi_i = o
\end{align*}

Next, we need to update \text{I}_{\text{fileSP}} \) so that it records that we have seen an open system call for all files that are currently open.

\[
\text{I}_{\text{fileSP}} \triangleq \lambda \text{st. file_auth} \; st * \exists i \in \tilde{I}. \; (\forall i \in \tilde{I}. \; \exists o. \; \text{in_trace} \; i \; o) * \forall z_f \in st. \; \exists i \in \tilde{I}. \; \text{in_trace} \; i \; (\text{open}, z_f)
\]

In this extended version, \text{I}_{\text{fileSP}} \) explicitly keeps track of a list \( \tilde{I} \) of valid indices into the trace, and ensures that for any currently open file handle \( z_f \), there exists some index \( i \in \tilde{I} \) such that the observation at that index in the trace witnesses the call to open that produced \( z_f \).
Now we can generalize the proof rule for syscall to its final form, which allows us to use the system call assumptions $A$.  

\[
P \text{-syscall-full} \quad \forall q_r \, s \, \pi. I_{SP} s \rightarrow P(s, q_a, \Phi) \triangleq \exists i. (\forall i \in \tilde{i}. \exists o. \text{in_trace} i o) \bigwedge \\
(\forall i_{new} > \max(\tilde{i}). \text{in_trace} i_{new} (s, q_a, q_r) \rightarrow \text{full_trace} \pi \rightarrow A \pi \implies \\
\exists \pi'. st \ (s, q_a, q_r)^\prime_{SP} \text{st}' \ast \text{full_trace} \pi \ast \Phi \ast q_r) \bigwedge \\
\{P(s, q_a, \Phi)\} \text{syscall}(s, q_a) \text{at High } \{q_r \text{ at High } \Phi \ast q_r\}\]

Let’s walk through this rule: After applying $P$-syscall-full, one learns that the system call invariant holds for the current state $(I_{SP} st)$. Then one has to provide a list of indices $\tilde{i}$ which have occurred in the trace $(\forall i \in \tilde{i}. \exists o. \text{in_trace} i o)$. In the proof for $T$-FILE-OPEN this is instantiated with $\tilde{i}$ from $I_{fileSP}$.

Afterwards one receives an index $i_{new}$ which is greater than all indices in $\tilde{i}$ (which implies $i_{new} \notin \tilde{i}$), and one knows that this index is now in the trace with the data of the current system call $(\text{in_trace} i_{new} (s, q_a, q_r))$. Furthermore, one gets the $\text{full_trace} \pi$ predicate and learns that the system call assumptions $A$ hold for the current trace $\pi$. The rest is the same as $P$-syscall except that one has to return ownership of $\text{full_trace} \pi$ for it to be available again when using $P$-syscall-full the next time.

The proof for $T$-FILE-OPEN uses the information from $P$-syscall-full in combination with the updated $I_{fileSP}$ and $A_{file}$ to conclude that the file handle $z_f$ returned by open is not in $st$. Then $P$-FILE-OPEN is used to create file_token $z_f$—which is used to establish the file corresponding to the file returned by $T$-FILE-OPEN—and the in_trace $i (s, q_a, q_r)$ provided by $P$-syscall-full is used to re-establish $I_{fileSP}$.

8 RELATED WORK

In Section 1, we already discussed related work on robust safety and sandboxing. In this section, we revisit related work on robust safety for a more technical comparison, and then discuss other related areas of research.

Robust safety. As mentioned in the introduction, many prior type systems and program logics enforce robust safety [Fournet et al. 2007; Bugliesi et al. 2011; Gordon and Jeffrey 2001; Swasey et al. 2017], but in abstract memory models where the untrusted code cannot guess locations that are not explicitly given to it. In contrast, we work with a low-level memory model where untrusted code can cast an integer to a pointer and dereference it. Sandboxing becomes relevant in this setting.

Another difference from prior work is the type at which the untrusted code is allowed to interact with verified code. In prior work, this type, called Un [Fournet et al. 2007; Bugliesi et al. 2011; Gordon and Jeffrey 2001] or Low [Swasey et al. 2017], includes only those values to which it is safe for untrusted code to have access, i.e., values from which it should not be possible to obtain access to a (high-privilege) location that trusted code has placed invariants on. This Un/Low type has a fairly complex semantic interpretation because it is not obvious from looking solely at a value whether or not one can use it to gain access to a high-privilege location. In contrast, in our setting, owing to physical memory isolation, the trusted and untrusted code can pass arbitrary surface values back and forth, and hence the analogue of the type Un/Low in our setting is simply the universal type, any.

Further differences arise in how safety properties are modeled. In our setting, we define safety properties in terms of traces of system calls. Prior approaches have used various different setups\footnote{Here the maximum of the empty list is defined as -1.}.
that fit their specific settings best. For example, Fournet et al. [2007] define safety in terms of sets of prior events, while Swasey et al. [2017] define safety as non-failure of inlined assertions.

**Automatic enforcement of types.** Enforcing types at the boundary of strongly typed and less strongly typed (or untyped) code has been extensively studied in the literature on decidable higher-order contracts [Findler and Felleisen 2002; Blume and McAllester 2004]. Contracts have even been examined for the interaction between substructural and standard types [Tov and Pucella 2010]. While inspired by such work, our wrappers go one step further as they not only check values, but also transform them. This allows us to implement and verify dynamic sealing in our framework, whereas it is built into the operational semantics of Tov and Pucella [2010]. Also, owing to our setting involving low-level untrusted code, the requirements on our wrappers are different — some of our wrappers also translate between different representations, e.g., the wrapper for $\Pi$.

**Gradual typing.** Interaction between typed and untyped code has also been studied in the literature on gradual typing [Siek and Taha 2006; Wadler and Findler 2009; Igarashi et al. 2017; Lehmann and Tanter 2017; New and Ahmed 2018]. In gradual typing, some parts of a program contain type annotations that are statically checked by the type checker, whereas other parts are not statically type-checked and, in those parts, types are enforced at run time. A value passed from an untyped part to a typed part is dynamically cast to the expected type. At a high level, our wrappers can be viewed as a replacement for such casts. However, the casts in gradual typing are usually expected to fulfill certain properties like restoration of the original value after it is passed in one direction and then back [New and Ahmed 2018]. We explicitly do not want this cancellativity property between import and export since it would disallow useful wrappers like Sealed$_s \tau$ where importing a seal created by export might fail (by design) if the seal was already used. A further difference is that our wrappers copy values between low and high heaps, which is not required in conventional gradual typing. Moreover, our types are semantically defined and thus extensible. Consequently, the set of wrappers (casts) is also extensible.

**Linear capabilities for fully abstract compilation of separation-logic-verified code.** In concurrent work, Van Strydonck et al. [2019] provide a fully abstract compiler from a language with separation logic specifications on untrusted functions to a capability machine with linear capabilities. The compiler inserts checks around the untrusted functions that fulfill a role similar to our wrappers. However, the implementations of these checks differ a great deal from our wrappers due to the different isolation techniques used (fine-grained linear capabilities in their work vs. coarse-grained memory isolation in ours). Besides, since Van Strydonck et al. [2019] prove full abstraction, their checks need to be both correct (i.e., programs distinguishable in the source language must be distinguishable in the target language after compilation) and secure (i.e., programs whose compilations are distinguishable in the target language must be distinguishable in the source language). The well-typedness requirement for our wrappers corresponds to the security direction. However, we do not require correctness, as this would rule out wrappers like dynamic sealing.

**Formalizing the security guarantees of compartmentalizing compilation.** There is existing work on formal, high-level properties of compartmentalization in the context of compilation. Juglaret et al. [2016] and Abate et al. [2018] investigate how a compiler can transfer properties of a compartmentalized source language application to the target language. While this work also relies on the notion of robust properties (at least superficially), it is actually complementary. Our paper focuses on establishing robust properties of the source application, whereas this line of work could be used to transfer those properties to the compiled program. Properly aligning our work and existing work to compose nicely in this way remains an open avenue for future research.
A OPERATIONAL SEMANTICS OF $\lambda_{\text{sandbox}}$

This appendix contains the complete operational semantics of $\lambda_{\text{sandbox}}$.

$$h \in \text{Heap} \triangleq \text{Loc}^{\text{fin}} \rightarrow \text{Val} \quad o \in \text{Observation} \triangleq (\text{Name}, \text{Lit}, \text{Lit}) \quad \pi \in \text{Trace} \triangleq \text{list Observation}$$

$$I \in \text{Interface} \triangleq (\text{Priv}, \text{Name})^{\text{fin}} \rightarrow \text{Val} \quad A \in \text{SyscallAssumptions} \triangleq \text{Trace} \rightarrow \text{Prop}$$

Like in $\lambda_{\text{Rust}}$ [Jung et al. 2018a], in $\lambda_{\text{sandbox}}$ two locations where at least one is not allocated can both compare as equal and as unequal.

$$h[l \leftarrow v] = h[l \leftarrow v] \quad \text{with the offset shifted by } z$$

In the rules for the operational semantics, $h[l \leftarrow v]$ denotes the heap $h$ with $l$ updated to $v$, and $h[l \leftarrow \bot]$ is the heap $h$ with the entry at location $l$ deleted. The notation $l + z$ denotes the location $l$ with the offset shifted by $z$.

$$\begin{array}{c}
\text{Low} \sqsubseteq \text{High} \\
\begin{array}{c}
\frac{h \vdash p_1 \equiv p_2}{h \vdash p_1 \sqsubseteq p_2}
\end{array}
\end{array}$$

$$\begin{array}{c}
\frac{h \vdash \lambda \[ l \leftarrow v \] \rightarrow \lambda \[ l \leftarrow v \] \quad l \neq l'}{h \vdash \lambda \[ l \leftarrow v \] \rightarrow \lambda \[ l \leftarrow v \]}
\end{array}$$

O-add

$$\frac{z_1 + z_2 = z'}{I \vdash h, \pi \mid z_1 + z_2 \rightarrow_A h, \pi \mid z' \at p}$$

O-sub

$$\frac{z_1 - z_2 = z'}{I \vdash h, \pi \mid z_1 - z_2 \rightarrow_A h, \pi \mid z' \at p}$$

O-le-true

$$\frac{z_1 \leq z_2}{I \vdash h, \pi \mid z_1 \leq z_2 \rightarrow_A h, \pi \mid 1 \at p}$$

O-le-false

$$\frac{z_1 > z_2}{I \vdash h, \pi \mid z_1 > z_2 \rightarrow_A h, \pi \mid 0 \at p}$$

O-eq-true

$$\frac{h \vdash q_1 = q_2}{I \vdash h, \pi \mid q_1 = q_2 \rightarrow_A h, \pi \mid 1 \at p}$$

O-eq-false

$$\frac{h \vdash q_1 \neq q_2}{I \vdash h, \pi \mid q_1 \neq q_2 \rightarrow_A h, \pi \mid 0 \at p}$$

O-offset

$$\frac{l' = l + z}{I \vdash h, \pi \mid l, z \rightarrow_A h, \pi \mid l' \at p}$$

O-app

$$\frac{I \vdash h, \pi \mid (\text{rec } f \langle \pi \rangle = e)(\pi \bar{\nu}) \rightarrow_A h, \pi \mid e[\text{rec } f \langle \pi \rangle = e/f, \pi/\bar{\nu}] \at p}{I \vdash h, \pi \mid \lambda \[ l \leftarrow v \] \rightarrow \lambda \[ l \leftarrow v \]}$$

O-read

$$\frac{h(l) = v}{I \vdash h, \pi \mid ! l \rightarrow_A h, \pi \mid v \at p}$$

O-write

$$\frac{h(l) = v' \quad l = (p', z_b, z_o) \quad p' \equiv p}{I \vdash h, \pi \mid l \leftarrow v \rightarrow_A h[l \leftarrow v], \pi \mid \text{write} \at p}$$

O-alloc

$$\frac{z > 0 \quad l = (p', z_b, z_o) \quad \{ (p', z_b) \} \times \mathbb{Z} \# \text{dom}(h) \quad p' \equiv p}{I \vdash h, \pi \mid \text{alloc}_{\text{p}}(z) \rightarrow_A h[l + m \leftarrow \text{write} \mid 0 \leq m < z], \pi \mid l \at p}$$
The High-Level Benefits of Low-Level Sandboxing

\[ z > 0 \]
\[ l = (p', z_b, z_o) \quad \text{dom}(h) \cap \{(p', z_b)\} \times \mathbb{Z} = \{(p', z_b)\} \times \{m|z_o \leq m < z_o + z\} \quad p' \subseteq p \]
\[ I \vdash h, \pi \mid \text{free}(z, l) \text{ at } p \rightarrow_A h[l + m \leftarrow \perp | 0 \leq m < z], \pi \mid \triangleright at p \]

\[ I \vdash h, \pi \mid \text{CAS}(l, q, v) \text{ at } p \rightarrow_A h, \pi \mid 0 \text{ at } p \]

\[ I \vdash h, \pi \mid \text{CAS}(l, q, v) \text{ at } p \rightarrow_A h[l \leftarrow v], \pi \mid 1 \text{ at } p \]

\[ I \vdash h, \pi \mid \text{SYSCALL} \]

\[ I \vdash h, \pi \mid \text{gated_call}(s, \overline{v}) \text{ at } p \rightarrow_A h, \pi \mid \text{gated_ret}(v, \overline{v}) \text{ at } p' \]

\[ I \vdash h, \pi \mid \text{get_offset} \text{ at } p \rightarrow_A h, \pi \mid v \text{ at } p' \]

\[ I \vdash h, \pi \mid \text{mk_location}(z, q, z_o) \text{ at } p \rightarrow_A h, \pi \mid l \text{ at } p \]

We wish to thank David Swasey and Ralf Jung for their feedback and helpful discussions. This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project “RustBelt”, funded under the European Union’s Horizon 2020 Framework Programme (grant agreement no. 683289) and by the German Science Foundation (DFG) through the Collaborative Research Center “Methods and Tools for Understanding and Controlling Privacy” (SFB 1223).

REFERENCES


