The High-Level Benefits of Low-Level Sandboxing

MICHAEL SAMMLER, MPI-SWS and Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
DEEPAK GARG, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany
TADEUSZ LITAK, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

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:
Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The High-Level Benefits of Low-Level Sandboxing. *Proc. ACM Program. Lang.* 4, POPL, Article 32 (January 2020), 32 pages. https://doi.org/10.1145/3371100

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

Authors' addresses: Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany; Deepak Garg, MPI-SWS, Saarland Informatics Campus, Germany, dg@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Tadeusz Litak, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, tadeusz.litak@fau.de.

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.

Relation 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 $\lambda_{\mathsf{sandbox}}$) and its operational semantics.
- After introducing a rich affine type system for $\lambda_{\mathsf{sandbox}}$ in Section 4, we show formally that sandboxing yields a principle for robust verification of safety properties of a typed $\lambda_{\mathsf{sandbox}}$

```
main ≜ λ _.                          main_sandbox ≜ λ _.
                                         fork {gated_call(untrusted_main, ())};
      let l = alloc(1);                  let l = alloc(1);
      l ← 42;                            l ← 42;
      syscall(unsafe_print, !l)          syscall(unsafe_print, !l)
```

Fig. 1. Example applications without and with calling untrusted code.

program. Specifically, it is enough to verify (via typing) only the trusted compartment of a program, so long as it assumes nothing about the values it is passed from the untrusted compartment. Section 5 shows that every safety property we establish of the trusted compartment holds while it inter-operates with anything in the untrusted compartment—even code that can guess arbitrary memory locations.

- We introduce an extensible, verified system of wrappers between trusted and untrusted code that allows *typed* interaction with untrusted code. While these type-directed wrappers are not essential for security, they significantly simplify the integration of the untrusted code into a typed, trusted application. Section 6 introduces the system for wrapping and shows several examples of wrappers of varying complexity.
- In Section 7, we briefly justify the soundness of our formal development by describing a semantic model for the type system.

We conclude in Section 8 with a discussion of related work. All our technical work is fully mechanized in Coq on top of the Iris framework for higher-order concurrent separation logic [Jung et al. 2018b]. The Coq development is available online at [Sammler et al. 2019].

2 KEY IDEAS

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, main$_{\text{sandbox}}$, 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
$\mathsf{untrusted\_main}_1 \triangleq \lambda\,\_.\ \texttt{syscall}(\mathsf{unsafe\_print}, 1000)$
$\mathsf{untrusted\_main}_2 \triangleq \lambda\,\_.\ \texttt{let}\ l = \mathsf{guess\_location}();\ l \leftarrow 1000$

Fig. 2. Malicious instantiations of the untrusted code.

directly invoke unsafe_print with an argument greater than 999, as illustrated by the instance $\mathsf{untrusted\_main}_1$ in Figure 2. Alternatively, it could *guess* the address of the location l and write a value greater than 999 to it, causing the unsafe_print in the body of $\mathsf{main}_{\mathsf{sandbox}}$ to be called with an argument greater than 999. This is illustrated by the instance $\mathsf{untrusted\_main}_2$ 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 $\mathsf{main}_{\mathsf{sandbox}}$ maintains its safety property no matter what the untrusted function untrusted_main is. One way to attain this property is to *sandbox* the execution of untrusted_main, *i.e.*, to contain or limit its effects. The primitive `gated_call` that wraps 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 $\mathsf{main}_{\mathsf{sandbox}}$ robustly safe, *i.e.*, safe against *any* untrusted_main function provided by the untrusted code? (2) Given these properties of the sandbox, how can we prove that $\mathsf{main}_{\mathsf{sandbox}}$ 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 $\mathsf{untrusted\_main}_2$ by preventing this function from writing l. (b) *System call isolation:* To rule out attacks like that in $\mathsf{untrusted\_main}_1$, 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 **any** containing all values at which trusted code can interact with untrusted code.

Wrappers. While system call isolation (with memory isolation) ensures robust safety, it can be very restrictive for untrusted code as it disallows all access to system calls. In practice, we may

[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.

$$\mathsf{checked\_print} \triangleq \lambda\, x.\ \texttt{if}\ x < 1000\ \texttt{then}\ \texttt{syscall}(\mathsf{unsafe\_print}, x)\ \texttt{else}\ \texttt{stuck}$$

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** $\triangleq \{z \mid z < 1000\}$.[2] Then, a programmer could instead write the following function:

$$\mathsf{typed\_print} \triangleq \lambda\, x : \mathbf{LessThan1000}.\ \texttt{syscall}(\mathsf{unsafe\_print}, x)$$

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 $\tau.\mathsf{import} : \mathbf{any} \to \tau$ and $\tau.\mathsf{export} : \tau \to \mathbf{any}$ 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:

$$\mathsf{wrap}\ (v : \tau \to \tau') \triangleq \lambda\, x.\ (\tau'.\mathsf{export})(v(\tau.\mathsf{import}(x))) : \mathbf{any} \to \mathbf{any}$$

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

$$\mathbf{LessThan1000}.\mathsf{import} \triangleq \lambda\, x.\ \texttt{if}\ x < 1000\ \texttt{then}\ x\ \texttt{else}\ \texttt{stuck} : \mathbf{any} \to \mathbf{LessThan1000}$$

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.

[2]The notation and idea for this type are inspired by work on refinement types [Freeman and Pfenning 1991; Xi and Pfenning 1998].

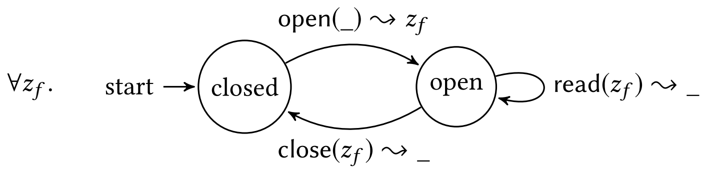

Fig. 3. System call policy fileSP for file system calls.

$$\begin{aligned}
\text{do\_open} &\triangleq \lambda\, n.\ \texttt{syscall}(\text{open}, n) : \mathbf{int} \to \mathbf{file}\\
\text{do\_read} &\triangleq \lambda\, f.\ \texttt{let}\ r = \texttt{syscall}(\text{read}, f);\ (f, r) : \mathbf{file} \to \Pi(\mathbf{file}, \mathbf{any})\\
\text{do\_close} &\triangleq \lambda\, f.\ \texttt{syscall}(\text{close}, f) : \mathbf{file} \to \mathbf{any}
\end{aligned}$$

Fig. 4. Functions shared with untrusted code for file system example.

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.

$$\begin{aligned}
&\qquad m \text{ is a global table that maps instances of } \textbf{file} \text{ to integer file handles:}\\
&\textbf{file}.\mathrm{import} \triangleq \lambda\, x.\ \texttt{if}\ x \in m\ \texttt{then}\ \mathrm{get\_and\_remove}(m, x)\ \texttt{else}\ \texttt{stuck} : \textbf{any} \longrightarrow \textbf{file}\\
&\textbf{file}.\mathrm{export} \triangleq \lambda\, x.\ \texttt{let}\ k = \mathrm{fresh\_key}(m);\ \mathrm{put}(m, k, x);\ k : \textbf{file} \longrightarrow \textbf{any}
\end{aligned}$$

Fig. 5. Wrappers for **file**.

$$\begin{aligned}
z &\in \mathbb{Z}\\
s &\in \mathit{Name}\\
\mathit{Priv} \ni p &::= \textbf{Low} \mid \textbf{High}\\
\mathit{Loc} \ni l &::= (p, z_b, z_o)\\
\mathit{Lit} \ni q &::= ☣ \mid z \mid l\\
\mathit{Val} \ni v &::= q \mid \texttt{rec}\ f\ (x) = e\\
\mathit{Expr} \ni e &::= x \mid v \mid e + e \mid e - e \mid e \leq e \mid e = e \mid e.e \mid e\,(\overline{e}) \mid\ !e \mid e \leftarrow e\\
&\quad \mid \texttt{CAS}(e_0, e_1, e_2) \mid \texttt{alloc}_p(e) \mid \texttt{free}(e_0, e_1) \mid \texttt{case}\ e\ \texttt{of}\ \overline{e} \mid \texttt{fork}\ e\\
&\quad \mid \texttt{mk\_location}(e_p, e_b, e_o) \mid \texttt{get\_priv}(e) \mid \texttt{get\_block}(e) \mid \texttt{get\_offset}(e)\\
&\quad \mid \texttt{syscall}(s, e) \mid \texttt{gated\_call}(s, \overline{e}) \mid \texttt{gated\_ret}(e)\\
\mathit{SurfaceExpr} \ni e &::= \{e \in \mathit{Expr} \mid e \text{ does not contain } \texttt{gated\_ret}\}\\
\mathit{SurfaceVal} \ni v &::= \{v \in \mathit{Val} \mid v \text{ does not contain } \texttt{gated\_ret}\}\\
\mathit{Ectx} \ni K &::= \text{(standard left-to-right evaluation contexts)}
\end{aligned}$$

Fig. 6. Definition of λ_{sandbox}.

$$\begin{aligned}
\lambda\,\overline{x}.\ e &\triangleq \texttt{rec}\ \_\ (\overline{x}) = e & \texttt{let}\ x = e_1;\ e_2 &\triangleq (\lambda\, x.\ e_2)\,(e_1)\\
e_1;\ e_2 &\triangleq \texttt{let}\ \_ = e_1;\ e_2 & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 &\triangleq \texttt{case}\ e_1\ \texttt{of}\ e_2, e_3\\
\texttt{stuck} &\triangleq 0\,(0)
\end{aligned}$$

Fig. 7. Standard derived forms.

3 LANGUAGE AND OPERATIONAL SEMANTICS

Notation. We use $\overline{\text{overline notation}}$ 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 λ_{sandbox}, which at its core is a standard concurrent lambda calculus modeled after λ_{Rust} [Jung et al. 2018a]. Standard abbreviations are displayed in Figure 7. λ_{sandbox} features standard binary operations and functions with multiple arguments. The heap is block-based like in λ_{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 $\texttt{alloc}(z)$ allocates a fresh block of size z initialized to the poison value ☣[3], $\texttt{free}(z, l)$ frees the block at location l of size z, and $l.z$ adds z to the offset of the location l.

[3]As in λ_{Rust}, ☣ is a special value for uninitialized memory. All operations except storing and reading a poison value to and from memory get stuck.

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

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

$$\boxed{p_1 \sqsubseteq p_2}$$

$$p \sqsubseteq p \qquad\qquad \textbf{Low} \sqsubseteq \textbf{High}$$

$$\boxed{\mathcal{I} \vdash h, \pi \mid e \ \textbf{at}\ p \rightarrow_A h', \pi' \mid e_1' \ \textbf{at}\ p_1', (e_2' \ \textbf{at}\ p_2')^?}$$

$$\text{O-READ}\quad \frac{h(l) = v}{\mathcal{I} \vdash h, \pi \mid\ !l \ \textbf{at}\ p \rightarrow_A h, \pi \mid v \ \textbf{at}\ p}
\qquad
\text{O-WRITE}\quad \frac{h(l) = v' \qquad l = (p', z_b, z_o) \qquad p' \sqsubseteq p}{\mathcal{I} \vdash h, \pi \mid l \leftarrow v \ \textbf{at}\ p \rightarrow_A h[l \leftarrow v], \pi \mid ☣ \ \textbf{at}\ p}$$

$$\text{O-SYSCALL}\quad \frac{\textbf{High} \sqsubseteq p \qquad A\,(\pi +\!\!+ [(s, q_a, q_r)])}{\mathcal{I} \vdash h, \pi \mid \texttt{syscall}(s, q_a) \ \textbf{at}\ p \rightarrow_A h, \pi +\!\!+ [(s, q_a, q_r)] \mid q_r \ \textbf{at}\ p}$$

$$\text{O-GATED-CALL}\quad \frac{p = \textbf{High} \Leftrightarrow p' = \textbf{Low} \qquad \mathcal{I}((p', s)) = v}{\mathcal{I} \vdash h, \pi \mid \texttt{gated\_call}(s, \overline{v}) \ \textbf{at}\ p \rightarrow_A h, \pi \mid \texttt{gated\_ret}(v\ (\overline{v})) \ \textbf{at}\ p'}$$

$$\text{O-GATED-RET}\quad \frac{p = \textbf{High} \Leftrightarrow p' = \textbf{Low}}{\mathcal{I} \vdash h, \pi \mid \texttt{gated\_ret}(v) \ \textbf{at}\ p \rightarrow_A h, \pi \mid v \ \textbf{at}\ p'}$$

$$\text{O-MK-LOCATION}\quad \frac{p' = \textbf{Low} \Leftrightarrow z_p = 0 \qquad z_b \geq 0 \qquad l = (p', z_b, z_o)}{\mathcal{I} \vdash h, \pi \mid \texttt{mk\_location}(z_p, z_b, z_o) \ \textbf{at}\ p \rightarrow_A h, \pi \mid l \ \textbf{at}\ p}$$

$$\text{O-GET-PRIV}\quad \frac{l = (p', z_b, z_o) \qquad p' = \textbf{Low} \Rightarrow z_p = 0 \qquad p' = \textbf{High} \Rightarrow z_p = 1}{\mathcal{I} \vdash h, \pi \mid \texttt{get\_priv}(l) \ \textbf{at}\ p \rightarrow_A h, \pi \mid z_p \ \textbf{at}\ p}$$

$$\text{O-GET-BLOCK}\quad \frac{l = (p', z_b, z_o)}{\mathcal{I} \vdash h, \pi \mid \texttt{get\_block}(l) \ \textbf{at}\ p \rightarrow_A h, \pi \mid z_b \ \textbf{at}\ p}$$

$$\text{O-GET-OFFSET}\quad \frac{l = (p', z_b, z_o)}{\mathcal{I} \vdash h, \pi \mid \texttt{get\_offset}(l) \ \textbf{at}\ p \rightarrow_A h, \pi \mid z_o \ \textbf{at}\ p}$$

$$\boxed{\mathcal{I} \vdash h, \pi \mid E \longrightarrow_A h', \pi' \mid E'}$$

$$\text{O-THREAD}\quad \frac{\mathcal{I} \vdash h, \pi \mid e \ \textbf{at}\ p \rightarrow_A h', \pi' \mid e_1' \ \textbf{at}\ p_1', (e_2' \ \textbf{at}\ p_2')^?}{\mathcal{I} \vdash h, \pi \mid E +\!\!+ [K[e] \ \textbf{at}\ p] +\!\!+ E' \longrightarrow_A h', \pi' \mid E +\!\!+ [K[e_1'] \ \textbf{at}\ p_1'] +\!\!+ E' +\!\!+ [e_2' \ \textbf{at}\ p_2']}$$

Fig. 8. Selected rules from the operational semantics of λ_{sandbox}.

As motivated in the previous sections, we want to model strong adversaries that can guess arbitrary locations. Thus, $\lambda_{\mathsf{sandbox}}$ provides the `mk_location`, `get_priv`, `get_block`, and `get_offset` instructions, which allow composing/splitting a location from/into its parts. Using `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

$$\mathcal{I} \vdash h, \pi \mid e \textbf{ at } p \rightarrow_A h', \pi' \mid e_1' \textbf{ at } p_1', (e_2' \textbf{ at } p_2')^?$$

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 π can result (after one step) in e_1' at privilege level p_1' with potentially modified heap h' and trace π'. Only the rule for `fork` uses the optional second resulting expression $(e_2' \textbf{ at } p_2')^?$ to signify that it spawned a new thread. The functions that represent the interface between the trusted and untrusted code are contained in $\mathcal{I}$.

The main reason for including concurrency in $\lambda_{\mathsf{sandbox}}$ 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.

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 **at High** states that e is executing outside of the sandbox. Only the trusted code should run at **High** privilege. Likewise, e **at Low** signifies that e runs inside the sandbox; untrusted code should always run at **Low** privilege.

To switch the privilege level, $\lambda_{\mathsf{sandbox}}$ provides a *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, *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 $\lambda_{\mathsf{sandbox}}$ using the `gated_call` and `gated_ret` instructions. The preregistered functions are stored in the *interface*

$$\mathcal{I} : (\mathit{Priv}, \mathit{Name}) \xrightarrow{\text{fin}} \mathit{Val}$$

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

The `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 *surface expression* (*surface value*) is an expression (value) that does not contain a `gated_ret`.

Memory isolation. Now we look at how $\lambda_{\mathsf{sandbox}}$ uses the privilege mechanism to prohibit the untrusted code from overriding memory of the trusted code. $\lambda_{\mathsf{sandbox}}$ models coarse-grained memory isolation by splitting the heap into a *high heap* of the trusted code and a *low heap* of the untrusted code via a privilege bit p in each location. A location that points to the high heap (*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 $p = \mathbf{Low}$ 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 $\texttt{alloc}_p$ 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.*, $p' \sqsubseteq p$. 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 $\lambda_{\mathsf{sandbox}}$'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, $\lambda_{\mathsf{sandbox}}$ contains a `syscall` instruction, which is used to define well-behaved programs.

As O-SYSCALL shows, system calls extend the trace $\pi : \mathit{list\ Observation}$ with the observation $o : (\mathit{Name}, \mathit{Lit}, \mathit{Lit})$ 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* $A : \mathit{Trace} \rightarrow \mathit{Prop}$, 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_{file}:[6]

$$A_{\mathsf{file}} \triangleq \lambda\pi.\ \forall i_1, i_2, q_{a1}, q_{a2}, q_{r1}, q_{r2}.\\ \pi_{i_1} = (\mathsf{open}, q_{a1}, q_{r1}) \wedge \pi_{i_2} = (\mathsf{open}, q_{a2}, q_{r2}) \wedge i_1 \neq i_2 \rightarrow q_{r1} \neq q_{r2}$$

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 $\lambda_{\mathsf{sandbox}}$ 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 $st^{init} \in S$, and the transition function $st \xrightarrow{o}{}^{?} st'$, which states that, upon

[5] See Appendix A.

[6] Actually, A_{file} also needs to ensure that open only returns integers, but we omit this here.

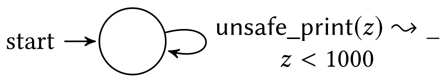

Fig. 9. System call policy printSP for the unsafe_print system call.

seeing the observation o, the automaton in state $st \in S$ steps to the state $st' \in S^{\perp}$ (*i.e.*, S extended with a "bad" state $\perp \notin S$). The automaton cannot leave the state $\perp$, and any state except the bad state $\perp$ 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 $s(q_a) \rightsquigarrow q_r$ denotes the observation (s, q_a, q_r). All transitions not depicted in the diagram implicitly go to the non-accepting $\perp$ state, which is also implicit.

We write $st \xrightarrow{o}^{\checkmark} st'$ to assert that $st \xrightarrow{o}^{?} st'$ and st' is not $\perp$. This can be extended to the judgment $st \xrightarrow{\pi}^{\checkmark} st'$ on traces, which runs the automaton on the observations o in the trace π using $\xrightarrow{o}^{\checkmark}$. This is used in the statement of the soundness theorem of the type system in Section 5, which shows that all traces π 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 λ_{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 λ_{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 λ_{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, $v \triangleleft^{SP,A} \tau$, states that the surface value v has the type τ. 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 v, *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:

$$T^{SP,A} ::= \emptyset \mid T^{SP,A}, v \triangleleft^{SP,A} \tau$$

The typing judgment for expressions

$$T_1 \vDash_p^{SP,A} e : \tau \mid T_2$$

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 v, then the context for the continuation of e is context $v \triangleleft^{SP,A} \tau, T_2$.

$$\textsc{T-weaken}\ \frac{T_1 \vDash_p e : \tau \mid T_2}{T_1, v \triangleleft \tau \vDash_p e : \tau \mid T_2} \qquad \textsc{T-copy}\ \frac{\tau \text{ copy} \qquad T_1, v \triangleleft \tau, v \triangleleft \tau \vDash_p e : \tau \mid T_2}{T_1, v \triangleleft \tau \vDash_p e : \tau \mid T_2} \qquad \textsc{T-val}\ \frac{v \triangleleft \tau}{\emptyset \vDash_p v : \tau \mid \emptyset}$$

$$\textsc{T-stuck}\ \frac{e \text{ is stuck}}{\emptyset \vDash_p e : \tau \mid T} \qquad \textsc{T-let}\ \frac{T_1 \vDash_p e_1 : \tau_1 \mid T_2 \qquad \forall v.\ v \triangleleft \tau_1, T_2, T \vDash_p e_2[v/x] : \tau_2 \mid T_3}{T_1, T \vDash_p \texttt{let}\, x = e_1;\ e_2 : \tau_2 \mid T_3}$$

Fig. 10. Structural typing rules.

$$\textsc{T-any-copy}\ \ \textbf{any} \text{ copy} \qquad \textsc{T-any}\ \ v \triangleleft \textbf{any} \qquad \textsc{T-int-copy}\ \ \textbf{int} \text{ copy} \qquad \textsc{T-int}\ \ z \triangleleft \textbf{int} \qquad \textsc{T-add}\ \ v_1 \triangleleft \textbf{any}, v_2 \triangleleft \textbf{any} \vDash_{\textbf{High}} v_1 + v_2 : \textbf{int} \mid \emptyset$$

$$\textsc{T-alloc}\ \frac{|\overline{\textbf{any}}| = n}{\emptyset \vDash_{\textbf{High}} \texttt{alloc}_{\textbf{High}}(n) : \Pi(\overline{\textbf{any}}) \mid \emptyset} \qquad \textsc{T-free}\ \frac{|\overline{\tau}| = n}{v \triangleleft \Pi(\overline{\tau}) \vDash_{\textbf{High}} \texttt{free}(n, v) : \textbf{any} \mid \emptyset}$$

$$\textsc{T-write}\ \ v_w \triangleleft \tau_w, v_p \triangleleft \Pi(\overline{\tau_p}) \vDash_{\textbf{High}} v_p.n \leftarrow v_w : \textbf{any} \mid v_p \triangleleft \Pi(\overline{\tau_p}[n \leftarrow \tau_w])$$

$$\textsc{T-read-copy}\ \frac{\tau_n \text{ copy}}{v \triangleleft \Pi(\overline{\tau}) \vDash_{\textbf{High}} !v.n : \tau_n \mid v \triangleleft \Pi(\overline{\tau})} \qquad \textsc{T-read}\ \ v \triangleleft \Pi(\overline{\tau}) \vDash_{\textbf{High}} !v.n : \tau_n \mid v \triangleleft \Pi(\overline{\tau}[n \leftarrow \textbf{any}])$$

$$\textsc{T-fn-copy}\ \ \textbf{fn}\ \overline{\tau_a} \longrightarrow \tau_r \text{ copy} \qquad \textsc{T-fn}\ \frac{T \text{ copy} \qquad \forall v_f\ \overline{v_a}.\ T, v_f \triangleleft \textbf{fn}\ \overline{\tau_a} \longrightarrow \tau_r, \overline{v_a \triangleleft \tau_a} \vDash_{\textbf{High}} e[v_f/f, \overline{v_a}/\overline{x}] : \tau_r \mid \emptyset}{T \vDash_{\textbf{High}} \texttt{rec}\, f\ (\overline{x}) = e : \textbf{fn}\ \overline{\tau_a} \longrightarrow \tau_r \mid \emptyset}$$

$$\textsc{T-call}\ \ v_f \triangleleft \textbf{fn}\ \overline{\tau_a} \longrightarrow \tau_r, \overline{v_a \triangleleft \tau_a} \vDash_{\textbf{High}} v_f\ (\overline{v_a}) : \tau_r \mid \emptyset$$

Fig. 11. Selected typing rules for the trusted code, $\overline{\tau}[n \leftarrow \tau']$ denotes $\overline{\tau}$ with τ_n replaced by τ'.

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 τ copy to mean that τ 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 $\forall v$ in the second premise is meta-level quantification. It means that the judgment after it is parametric in v.

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.[7] 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 z.) 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**.

λ_{sandbox} models a low-level language so it does not have native products. Instead products are emulated using the heap. The product type $\Pi(\overline{\tau})$ is a pointer to a block of $|\overline{\tau}|$ values, where the value at position n has the type τ_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 τ_n to the type of the written value τ_w, as T-WRITE shows. There are two rules for reading from a product. If the read type τ_n is copyable, it is possible to use T-READ-COPY, which keeps the product unchanged as one would expect. However, if τ_n is not copyable, τ_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 $\mathbf{fn}\,\overline{\tau_a} \to \tau_r$. Rule T-FN says that a value is a well-typed function of type $\mathbf{fn}\,\overline{\tau_a} \to \tau_r$ if the body of the function e substituted with values $\overline{v_a}$ of the argument types τ_a is a well-typed expression of the return type τ_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_{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:

$$\text{T-GATED-CALL} \qquad \overline{v \triangleleft \mathbf{any}} \vDash_{\mathbf{High}} \texttt{gated\_call}(s, \overline{v}) : \mathbf{any} \mid \emptyset$$

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

[7]Technically, **any** is inhabited only by all *surface values*, but this is implicit since $v \triangleleft \tau$ is only defined for surface values v.

$$\textsc{T-lessthan1000}\quad \frac{z < 1000}{z \triangleleft \mathbf{LessThan1000}}$$

$$\textsc{T-print}\quad v \triangleleft \mathbf{LessThan1000} \vDash^{\mathsf{printSP}}_{\mathbf{High}} \mathtt{syscall}(\text{unsafe\_print}, v) : \mathbf{any} \mid \emptyset$$

$$\textsc{T-file-open}\quad v \triangleleft \mathbf{int} \vDash^{\mathsf{fileSP}, A_{\mathsf{file}}}_{\mathbf{High}} \mathtt{syscall}(\text{open}, v) : \mathbf{file} \mid \emptyset$$

$$\textsc{T-file-read}\quad v \triangleleft \mathbf{file} \vDash^{\mathsf{fileSP}, A_{\mathsf{file}}}_{\mathbf{High}} \mathtt{syscall}(\text{read}, v) : \mathbf{any} \mid v \triangleleft \mathbf{file}$$

$$\textsc{T-file-close}\quad v \triangleleft \mathbf{file} \vDash^{\mathsf{fileSP}, A_{\mathsf{file}}}_{\mathbf{High}} \mathtt{syscall}(\text{close}, v) : \mathbf{any} \mid \emptyset$$

Fig. 12. Typing rules for printSP and fileSP.

of type **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 [Sammler et al. 2019] 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-GATED-CALL by proving in Theorem 2 that any surface expression is semantically well-typed at **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 `gated_call`.

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

Theorem 2 (Soundness of the sandbox)**.** *For all surface expressions e, the following rule is sound:*

$$\emptyset \vDash_{\mathbf{Low}} e : \mathbf{any} \mid \emptyset$$

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 $\mathbf{fn}\ \overline{\mathbf{any}} \rightarrow \mathbf{any}$.

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 `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 $\mathcal{I}$, one must show that its application to the supplied arguments $\overline{v}$ 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* st_{SP}^{init},
- *any system call assumptions A,*
- *any map from block names to initial values of global variables* $m : Name \xrightarrow{fin} list\ Lit$,
- *any value main, and*
- *any initial context* T_{init},

the following rule is sound:

$$\frac{\begin{array}{cc} (1)\ T_{init}\ copy & (2)\ \mathit{InitFromHeap}(m, T_{init}) \\ (3)\ T_{init} \vDash^{SP,A}_{\mathbf{High}} main : \mathbf{fn}\,[\,] \to \mathbf{any} \mid \emptyset & (4)\ \forall v \in \overline{v_t}.\ T_{init} \vDash^{SP,A}_{\mathbf{High}} v : \mathbf{fn}\ \overline{\mathbf{any}} \to \mathbf{any} \mid \emptyset \\ (5)\ \mathcal{I} = [\overline{(\mathbf{Low}, s_u) \mapsto v_u}, \overline{(\mathbf{High}, s_t) \mapsto v_t}] & (6)\ \mathcal{I} \vdash h_{init}\, m, [\,] \mid main\,()\ \mathbf{at\ High} \longrightarrow^*_A h, \pi \mid E \end{array}}{(7)\ \exists st.\ st^{init}_{SP} \xrightarrow{\pi}{}^{\checkmark}_{SP} st}$$

The main goal of this theorem is to show that every trace π 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 $g2l : Name \to Loc$, 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}. $\mathsf{InitFromHeap}(m, T_{init})$ 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 **any** type. However, the **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 $\mathbf{fn}\ \overline{\tau_a} \to \tau_r$ and *automatically* lift it to a function of the type $\mathbf{fn}\ \overline{\mathbf{any}} \to \mathbf{any}$, which can be shared with untrusted code. Dually, it allows an untrusted function of type $\mathbf{fn}\ \overline{\mathbf{any}} \to \mathbf{any}$ to be lifted automatically to the type $\mathbf{fn}\ \overline{\tau_a} \to \tau_r$, which can then be applied safely at that type by trusted code.

This lifting mechanism works through a system of *wrappers* converting values to (or from) the **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 τ:

- τ.import $\lhd$ $\mathbf{fn\ any} \to \tau$: A type τ 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 *importable*. Simple types like **LessThan1000** from Section 2 can be imported by checking if the untrusted code passed a value of the type τ and either passing the value to the trusted code or getting stuck.
- τ.export $\lhd$ $\mathbf{fn}\ \tau \to \mathbf{any}$: By providing an export function a type τ 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 *exportable*. It is always possible to implement τ.export as the identity function since **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 Import and 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 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 τ_a.export function on each argument to transform all values in $\overline{x}$ to **any**. Then wrap_import calls the untrusted code using `gated_call`. This call is safe due to T-GATED-CALL and returns a value of type **any**, which is checked and transformed to a value of type τ_r using τ_r.import. In total, this means that wrap_import can be typed at type $\mathbf{fn}\ \overline{\tau_a} \to \tau_r$, as Figure 13 shows.

$$\text{wrap\_import}\ s\ \overline{\tau_a}\ \tau_r \triangleq \lambda\,\overline{x}.\ \overline{\texttt{let}\ x = \tau_a.\text{export}(x);}\ \texttt{let}\ r = \texttt{gated\_call}(s, \overline{x});\ \tau_r.\text{import}\,(r)$$
$$\text{wrap\_export}\ v\ \overline{\tau_a}\ \tau_r \triangleq \lambda\,\overline{x}.\ \overline{\texttt{let}\ x = \tau_a.\text{import}(x);}\ \texttt{let}\ r = v\,(\overline{x});\ \tau_r.\text{export}\,(r)$$

T-WRAP-IMPORT
$$\frac{T_r \vDash_{\textbf{High}} \tau_r.\text{import} : \textbf{fn any} \to \tau_r \mid \emptyset \qquad \overline{T_a}, T_r\ \text{copy} \qquad \overline{T_a \vDash_{\textbf{High}} \tau_a.\text{export} : \textbf{fn}\ \tau_a \to \textbf{any} \mid \emptyset}}{\overline{T_a}, T_r \vDash_{\textbf{High}} \text{wrap\_import}\ s\ \overline{\tau_a}\ \tau_r : \textbf{fn}\ \overline{\tau_a} \to \tau_r \mid \emptyset}$$

T-WRAP-EXPORT
$$\frac{\overline{T_a}, T_v, T_r\ \text{copy} \qquad T_v \vDash_{\textbf{High}} v : \textbf{fn}\ \overline{\tau_a} \to \tau_r \mid \emptyset \qquad \overline{T_a \vDash_{\textbf{High}} \tau_a.\text{import} : \textbf{fn any} \to \tau_a \mid \emptyset} \qquad T_r \vDash_{\textbf{High}} \tau_r.\text{export} : \textbf{fn}\ \tau_r \to \textbf{any} \mid \emptyset}{\overline{T_a}, T_v, T_r \vDash_{\textbf{High}} \text{wrap\_export}\ v\ \overline{\tau_a}\ \tau_r : \textbf{fn}\ \overline{\textbf{any}} \to \textbf{any} \mid \emptyset}$$

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

$$\begin{array}{ll} \textbf{any}.\text{import} \triangleq \lambda\,x.\ x \mathrel{\triangleleft} \textbf{fn any} \to \textbf{any} & \textbf{int}.\text{import} \triangleq \lambda\,x.\ x + 0 \mathrel{\triangleleft} \textbf{fn any} \to \textbf{int} \\ \textbf{any}.\text{export} \triangleq \lambda\,x.\ x \mathrel{\triangleleft} \textbf{fn any} \to \textbf{any} & \textbf{int}.\text{export} \triangleq \lambda\,x.\ x \mathrel{\triangleleft} \textbf{fn int} \to \textbf{any} \end{array}$$

Fig. 14. import and export wrappers for **any** and **int**.

T-LOW-COPY
$$\textbf{LowPtr}\ \text{copy}$$

T-ALLOC-LOW
$$\emptyset \vDash_p \texttt{alloc}_{\textbf{Low}}(z) : \textbf{LowPtr} \mid \emptyset$$

T-FREE-LOW
$$v \mathrel{\triangleleft} \textbf{LowPtr} \vDash_p \texttt{free}(z, v) : \textbf{any} \mid \emptyset$$

T-WRITE-LOW
$$v_w \mathrel{\triangleleft} \textbf{any}, v_p \mathrel{\triangleleft} \textbf{LowPtr} \vDash_p v_p.z \leftarrow v_w : \textbf{any} \mid \emptyset$$

T-READ-LOW
$$v \mathrel{\triangleleft} \textbf{LowPtr} \vDash_p\ !v.z : \textbf{any} \mid \emptyset$$

Fig. 15. Typing rules for **LowPtr**

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

We define wrap_import and wrap_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

$$\texttt{let}\ x_1 = \tau_{a,1}.\text{export}(x_1);\ \ \texttt{let}\ x_2 = \tau_{a,2}.\text{export}(x_2);\ \ldots$$

Examples of wrappers. Some of the simplest wrappers are the wrappers for **any** and **int**, which can be seen in Figure 14. The wrappers for **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 **int**.import $\triangleleft$ **fn any** $\to$ **int** by T-ADD. Exporting an **int** simply returns it unchanged to the untrusted code.

$$\begin{aligned}\mathbf{LowPtr}.\mathsf{import} &\triangleq \lambda\, x.\ \mathtt{if\ get\_priv}(x)\ \mathtt{then\ stuck\ else}\ x \triangleleft \mathbf{fn\ any} \longrightarrow \mathbf{LowPtr}\\ \mathbf{LowPtr}.\mathsf{export} &\triangleq \lambda\, x.\ x \triangleleft \mathbf{fn\ LowPtr} \longrightarrow \mathbf{any}\end{aligned}$$

Fig. 16. import and export wrappers for **LowPtr**.

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:

$$\textsc{T-low-ptr}\qquad \frac{l = (\mathbf{Low}, \_, \_)}{l \triangleleft \mathbf{LowPtr}}$$

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 $\Pi(\overline{\tau})$.

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 $\Pi(\overline{\tau})$ type can be seen in Figure 17. Its import reads $|\overline{\tau}|$ 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 $\Pi(\overline{\tau})$.import (and $\Pi(\overline{\tau})$.export) is a macro similar to wrap_import and wrap_export to support products of arbitrary size. The expression $l.i \longleftarrow \tau.\mathsf{import}(!u.i)$ is repeated for all types τ in $\overline{\tau}$ with i referring to the index of τ in $\overline{\tau}$.

The most obvious difference from the previous wrappers is that the $\Pi(\overline{\tau})$.export function is not just the identity. Products are implemented as allocations on the high heap. Simply passing a high

$$
\begin{array}{ll}
\Pi(\overline{\tau}).\mathsf{import} \triangleq \lambda\, x. & \Pi(\overline{\tau}).\mathsf{export} \triangleq \lambda\, x. \\
\quad \mathtt{let}\ u = \mathbf{LowPtr}.\mathsf{import}(x); & \quad \underline{\mathtt{let}\ l = \mathtt{alloc}_{\mathbf{Low}}(|\overline{\tau}|);} \\
\quad \underline{\mathtt{let}\ l = \mathtt{alloc}_{\mathbf{High}}(|\overline{\tau}|);} & \quad \overline{l.i \leftarrow \tau.\mathsf{export}(!x.i);} \\
\quad \overline{l.i \leftarrow \tau.\mathsf{import}(!u.i);} & \quad \mathtt{free}(|\overline{\tau}|, x); \\
\quad \mathtt{free}(|\overline{\tau}|, u); & \quad \mathbf{LowPtr}.\mathsf{export}(l) \\
\quad l &
\end{array}
$$

$$
\frac{\overline{T}\ \mathsf{copy} \qquad \overline{T \vDash_{\mathbf{High}} \tau.\mathsf{import} : \mathbf{fn}\ \mathbf{any} \longrightarrow \tau \mid \emptyset}}{\overline{T} \vDash_{\mathbf{High}} \Pi(\overline{\tau}).\mathsf{import} : \mathbf{fn}\ \mathbf{any} \longrightarrow \Pi(\overline{\tau}) \mid \emptyset}
$$

$$
\frac{\overline{T}\ \mathsf{copy} \qquad \overline{T \vDash_{\mathbf{High}} \tau.\mathsf{export} : \mathbf{fn}\ \tau \longrightarrow \mathbf{any} \mid \emptyset}}{\overline{T} \vDash_{\mathbf{High}} \Pi(\overline{\tau}).\mathsf{export} : \mathbf{fn}\ \Pi(\overline{\tau}) \longrightarrow \mathbf{any} \mid \emptyset}
$$

Fig. 17. import and export wrappers for $\Pi(\overline{\tau})$.

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, $\Pi(\overline{\tau})$.import uses the import function of $\tau \in \overline{\tau}$. This generic approach to wrapping means that the wrappers compose nicely and $\Pi(\overline{\tau})$ is importable (exportable) if all types in $\overline{\tau}$ are importable (exportable).

unsafe_print example. Now we have everything in place to revisit the unsafe_print example of Section 2 and show that $\mathsf{main}_{\mathsf{file}}$ function is safe to execute for arbitrary untrusted code $\overline{v_u}$ 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.*, $A \triangleq \lambda\, \pi.\ True$) results in the obligations

$$
\mathsf{main}_{\mathsf{sandbox}} \lhd^{\mathsf{printSP}, A}\ \mathbf{fn}\ [\,] \longrightarrow \mathbf{any} \qquad \mathsf{typed\_print} \lhd^{\mathsf{printSP}, A}\ \mathbf{fn}\ \mathbf{LessThan1000} \longrightarrow \mathbf{any}
$$

which are easy to show with the typing rules described so far. All in all, we get

$$
\frac{\begin{array}{c}\mathcal{I} = [\overline{(\mathbf{Low}, s_u) \mapsto v_u}, (\mathbf{High}, \mathsf{typed\_print}) \mapsto \mathsf{wrap\_export}\ [\mathbf{LessThan1000}]\ \mathbf{any}\ \mathsf{typed\_print}] \\ \mathcal{I} \vdash \emptyset, [\,] \mid \mathsf{main}_{\mathsf{sandbox}}\ ()\ \mathbf{at}\ \mathbf{High} \longrightarrow^{*}_{A} h, \pi \mid E\end{array}}{\exists st.\ st^{init}_{\mathsf{printSP}} \xrightarrow{\pi}{}^{\checkmark}_{\mathsf{printSP}}\ st}
$$

This says that if we execute $\mathsf{main}_{\mathsf{sandbox}}$ (), which invokes the untrusted_main function of the arbitrary untrusted code $\overline{v_u}$[8], which in turn may invoke the *wrapped* version of typed_print, then the trace π of that execution adheres to the system call policy printSP.

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

[8]If the untrusted code does not expose a function named untrusted_main, this call gets stuck, which is not a problem.

$$
\begin{aligned}
\textbf{Sealed}_{s,\tau}.\text{import} \triangleq \lambda\, x.\; & \\
& \texttt{let}\ \mathit{seal} = \textbf{int}.\text{import}(x); \\
& \texttt{let}\ \mathit{mutex\_map} = \text{acquire}(\mathit{g2l}\ s); \\
& \texttt{let}\ \mathit{map\_unsealed} = \text{map\_take}(!\mathit{mutex\_map}.1, \mathit{seal}); \\
& \text{release}(!\mathit{mutex\_map}.0, !\mathit{map\_unsealed}.0); \\
& \text{unwrap}(!\mathit{map\_unsealed}.1) \\
\textbf{Sealed}_{s,\tau}.\text{export} \triangleq \lambda\, x.\; & \\
& \texttt{let}\ \mathit{mutex\_map} = \text{acquire}(\mathit{g2l}\ s); \\
& \texttt{let}\ \mathit{map\_seal} = \text{map\_put}(!\mathit{mutex\_map}.1, x); \\
& \text{release}(!\mathit{mutex\_map}.0, !\mathit{map\_seal}.0); \\
& \textbf{int}.\text{export}(!\mathit{map\_seal}.1)
\end{aligned}
$$

T-SEALED-IMPORT
$$\mathit{g2l}\ s \triangleleft \textbf{mutex}_{\textbf{map}_\tau} \vDash_{\textbf{High}} \textbf{Sealed}_{s,\tau}.\text{import} : \textbf{fn any} \rightarrow \textbf{Sealed}_{s,\tau} \mid \emptyset$$

T-SEALED-EXPORT
$$\mathit{g2l}\ s \triangleleft \textbf{mutex}_{\textbf{map}_\tau} \vDash_{\textbf{High}} \textbf{Sealed}_{s,\tau}.\text{export} : \textbf{fn Sealed}_{s,\tau} \rightarrow \textbf{any} \mid \emptyset$$

Fig. 18. import and export wrappers for $\textbf{Sealed}_{s,\tau}$.

$\textbf{mutex}_\tau$ copy	
type	function
$\textbf{mutex}_\tau/\textbf{guard}_\tau$	acquire $\triangleleft$ $\textbf{fn mutex}_\tau \rightarrow \Pi(\textbf{guard}_\tau, \tau)$
	release $\triangleleft$ $\textbf{fn guard}_\tau,\ \tau \rightarrow \textbf{any}$
$\textbf{map}_\tau$	map_put $\triangleleft$ $\textbf{fn map}_\tau, \tau \rightarrow \Pi(\textbf{map}_\tau, \textbf{int})$
	map_take $\triangleleft$ $\textbf{fn map}_\tau, \textbf{int} \rightarrow \Pi(\textbf{map}_\tau, \textbf{option}_\tau)$
$\textbf{option}_\tau$	unwrap $\triangleleft$ $\textbf{fn option}_\tau \rightarrow \tau$

Fig. 19. Types and functions used by wrappers for $\textbf{Sealed}_{s,\tau}$.

code is to use dynamic sealing [Morris 1973]. In this approach, one passes an opaque handle instead of the object itself. This handle is passed back to the trusted code to invoke methods on the object. In the following we show how this pattern can be implemented in our framework in a generic way.

For this we define a new type $\textbf{Sealed}_{s,\tau}$. Block s stores the map necessary for the dynamic sealing as will be explained later. $\textbf{Sealed}_{s,\tau}$ is inhabited by the same values as τ. We employ the usual notation for bidirectional inference rules:

$$\text{T-SEALED}\quad \frac{v \triangleleft \tau}{v \triangleleft \textbf{Sealed}_{s,\tau}}$$

The important point about $\textbf{Sealed}_{s,\tau}$ is that it is importable and exportable for all types τ, even if τ 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 $\textbf{Sealed}_{s,\tau}$ can be treated as a value of type τ during verification and thus $\textbf{Sealed}_{s,\tau}$ does not influence the verification of the trusted code.

Figure 18 shows how $\textbf{Sealed}_{s,\tau}$ 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 $\mathbf{Sealed}_{s,\tau}$.

- $\mathbf{mutex}_\tau$ 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 $\mathbf{guard}_\tau$, which release uses to put 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 $\mathbf{mutex}_\tau$ is copyable independent of τ copy.
- $\mathbf{map}_\tau$ 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 $\mathbf{option}_\tau$ 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.

$\mathbf{Sealed}_{s,\tau}$ 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_{init}:

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

This means that we have to show

$$\frac{\mathcal{I} = \begin{bmatrix} \overline{(\mathbf{Low}, s_t) \mapsto v_u}, \\ (\mathbf{High}, \mathsf{do\_open}) \mapsto \mathsf{wrap\_export}\ [\mathbf{int}]\ \mathbf{Sealed}_{\mathsf{file},\mathbf{file}}\ \mathsf{do\_open}, \\ (\mathbf{High}, \mathsf{do\_read}) \mapsto \mathsf{wrap\_export}\ [\mathbf{Sealed}_{\mathsf{file},\mathbf{file}}]\ \Pi(\mathbf{Sealed}_{\mathsf{file},\mathbf{file}}, \mathbf{any})\ \mathsf{do\_read}, \\ (\mathbf{High}, \mathsf{do\_close}) \mapsto \mathsf{wrap\_export}\ [\mathbf{Sealed}_{\mathsf{file},\mathbf{file}}]\ \mathbf{any}\ \mathsf{do\_close} \end{bmatrix} \qquad \mathcal{I} \vdash h_{init}\, m, [\,] \mid \mathsf{main}_{\mathsf{file}}\ ()\ \mathbf{at\ High} \longrightarrow^*_{A_{\mathsf{file}}} h', \pi \mid E}{\exists st.\ st^{init}_{\mathsf{fileSP}} \xrightarrow{\pi}{}^{\checkmark}_{\mathsf{fileSP}}\ st}$$

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

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

After applying Theorem 3 and establishing the initial context

$$T_{init} \triangleq g2l\ \mathsf{file} \triangleleft \mathbf{mutex}_{\mathbf{map}_{\mathbf{file}}}$$

one can use T-WRAP-EXPORT together with the typing rules for the wrappers to type the exposed functions. T_{init} is used by T-SEALED-IMPORT and T-SEALED-EXPORT. Then it is only necessary to show the following:

$$\mathsf{main}_{\mathsf{file}} \triangleleft^{\mathsf{fileSP}, A_{\mathsf{file}}} \mathbf{fn}\ [\,] \to \mathbf{any} \qquad \mathsf{do\_read} \triangleleft^{\mathsf{fileSP}, A_{\mathsf{file}}} \mathbf{fn}\ \mathbf{Sealed}_{\mathsf{file},\mathbf{file}} \to \Pi(\mathbf{Sealed}_{\mathsf{file},\mathbf{file}}, \mathbf{any})$$

$$\mathsf{do\_open} \triangleleft^{\mathsf{fileSP}, A_{\mathsf{file}}} \mathbf{fn}\ \mathbf{int} \to \mathbf{Sealed}_{\mathsf{file},\mathbf{file}} \qquad \mathsf{do\_close} \triangleleft^{\mathsf{fileSP}, A_{\mathsf{file}}} \mathbf{fn}\ \mathbf{Sealed}_{\mathsf{file},\mathbf{file}} \to \mathbf{any}$$

The type of $\mathsf{main}_{\mathsf{file}}$ follows from T-GATED-CALL; the others are easy to show by first replacing $\mathbf{Sealed}_{\mathsf{file},\mathbf{file}}$ with $\mathbf{file}$ 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 λ_{sandbox} to attain the program logic Iris$_{\text{sandbox}}$. Like the type system, Iris$_{\text{sandbox}}$ and thus also the type of its propositions $iProp^{SP,A}_{\text{sandbox}}$ are parameterized by a system call policy SP and system call assumptions A.

Definition 4. A *type* τ is a predicate $[\![\tau]\!]^{SP,A} \in SurfaceVal \longrightarrow iProp^{SP,A}_{\text{sandbox}}$. We write $v \triangleleft^{SP,A} \tau$ for $[\![\tau]\!]^{SP,A}\, v$.[9]

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 $P * Q \Rightarrow P$ [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 **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 τ is copyable if $v \triangleleft \tau$ is freely duplicable. In the model this is realized by defining a type τ as copyable if $v \triangleleft \tau$ 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 $[\![T]\!]$ is defined as the separating conjunction of all $v \triangleleft \tau$ 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:

$$T_1 \vDash^{SP,A}_{p} e : \tau \mid T_2 \quad \triangleq \quad \{[\![T_1]\!]^{SP,A}\}\, e \,\mathbf{at}\, p\, \{v\, \mathbf{at}\, p.\ v \triangleleft^{SP,A} \tau * [\![T_2]\!]^{SP,A}\}_?$$

The Hoare triple asserts that e is safe to run at privilege p if $[\![T_1]\!]$ holds. If the execution of e terminates in a value v, the privilege will be p, v will have type τ, and $[\![T_2]\!]$ 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 **any** and **int** are straightforward as they are defined to be inhabited by all surface values and all integers, respectively:

$$[\![\mathbf{any}]\!] \triangleq \lambda\, v.\ True \qquad [\![\mathbf{int}]\!] \triangleq \lambda\, v.\ \exists z : \mathbb{Z}.\ v = z$$

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

[9] $v \triangleleft^{SP,A} \tau$ for values that are not surface values is defined as *False*.

the low heap, which is necessary to fulfill assumption (1) of Theorem 2. Using this predicate, $\Pi(\overline{\tau})$ is defined as follows:

$$[\![\Pi(\overline{\tau})]\!] \triangleq \lambda\, v.\ \exists l.\ v = l * \exists \overline{v_p}.\ l \mapsto \overline{v_p} * \overline{v_p \triangleleft \tau} * \mathrm{DeallocSize}(l, |\overline{\tau}|)$$

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

The type of functions $\mathbf{fn}\ \overline{\tau_a} \longrightarrow \tau_r$ is defined in the same way as in Jung et al. [2018a] and closely follows T-FN in Figure 11:[10]

$$[\![\mathbf{fn}\ \overline{\tau_a} \longrightarrow \tau_r]\!] \triangleq \lambda\, v.\ \square \exists f \overline{x} e.\ v = (\texttt{rec}\ f\ (\overline{x}) = e) * \forall \overline{v_a}.\ \overline{v_a \triangleleft \tau_a} \vDash_{\mathbf{High}} e[v/f, \overline{v_a}/\overline{x}] : \tau_r \mid \emptyset$$

The modality $\square$ ensures that only persistent resources are used in the proof of $v \triangleleft \mathbf{fn}\ \overline{\tau_a} \longrightarrow \tau_r$, 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:

$$[\![\mathbf{LowPtr}]\!] \triangleq \lambda\, v.\ \exists l.\ v = l * l = (\mathbf{Low}, \_, \_)$$

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:

$$I_{SP} : S_{SP} \longrightarrow iProp^{SP,A}_{\mathsf{sandbox}}$$

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

$$\dfrac{P(s, q_a, \Phi) \triangleq \forall q_r, st.\ \ I_{SP}\ st \Rrightarrow \exists st'.\ st \overset{(s,q_a,q_r)\checkmark}{\Rightarrow}_{SP} st' * I_{SP}\ st' * \Phi\ q_r}{\{P(s, q_a, \Phi)\}\ \mathbf{syscall}(s, q_a)\ \mathbf{at\ High}\ \{q_r\ \mathbf{at\ High}.\ \Phi\ q_r\}_?}\ \text{P-SYSCALL}$$

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 st of the system call policy, SP accepts s for all return values q_r by stepping to a new state $st' \neq \bot$ and I_{SP} holds for the new state st'.

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

[10]For purposes of presentation, we suppress the $\triangleright$ modality, which is necessary to prove T-FN for recursive functions.
[11]To a first approximation, readers unfamiliar with Iris can read the view shift $\Rrightarrow$ as if it were the magic wand $-\!\!*$.

by the following rules:[12]

P-FILE-OPEN

$$\frac{z_f \notin st}{\mathsf{file\_auth}\, st \Rrightarrow \mathsf{file\_auth}\,(st \cup \{z_f\}) * \mathsf{file\_token}\, z_f}$$

P-FILE-IN

$$\mathsf{file\_auth}\, st * \mathsf{file\_token}\, z_f \twoheadrightarrow z_f \in st$$

P-FILE-CLOSE

$$\mathsf{file\_auth}\, st * \mathsf{file\_token}\, z_f \Rrightarrow \mathsf{file\_auth}\,(st \setminus \{z_f\})$$

Conceptually, file_auth st asserts that the set of open files is currently st and file_token z_f states that z_f is in this set (as 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 file_token z_f, as can be seen in P-FILE-CLOSE. I_{fileSP} maintains file_auth st, and **file** is defined using file_token z_f:

$$I_{\mathsf{fileSP}} \triangleq \lambda\, st.\ \mathsf{file\_auth}\, st \qquad [\![\mathbf{file}]\!] \triangleq \lambda\, v.\ \exists z_f.\ v = z_f * \mathsf{file\_token}\, z_f$$

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

Unfortunately, the simplified version of P-SYSCALL stated above is not quite strong enough to prove T-FILE-OPEN. To use P-FILE-OPEN we must show that z_f is not already an open file. Otherwise we could use P-FILE-OPEN to generate arbitrarily many 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_{file} defined in Section 3. However, as stated above, P-SYSCALL does not make use of A_{file}, and thus to prove T-FILE-OPEN we will need a stronger version of P-SYSCALL.

Before giving the stronger P-SYSCALL-FULL rule, we first need to introduce two new kinds of ghost state for reasoning about the trace:

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

The important property of full_trace π and in_trace $i\ o$ is that they can be combined to learn $\pi_i = o$.

$$\text{P-IN-TRACE} \qquad \mathsf{full\_trace}\, \pi * \mathsf{in\_trace}\, i\, o \twoheadrightarrow \pi_i = o$$

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

$$I_{\mathsf{fileSP}} \triangleq \lambda\, st.\ \mathsf{file\_auth}\, st * \exists \bar{i}.\ (\forall i \in \bar{i}.\ \exists o.\ \mathsf{in\_trace}\, i\, o) * \forall z_f \in st.\ \exists i \in \bar{i}.\ \mathsf{in\_trace}\, i\, (\mathsf{open}, \_, z_f)$$

In this extended version, I_{fileSP} explicitly keeps track of a list $\bar{i}$ of valid indices into the trace, and ensures that for any currently open file handle z_f, there exists some index $i \in \bar{i}$ such that the observation at that index in the trace witnesses the call to open that produced z_f.

[12]The definition of file_auth st and file_token z_f and the proofs of their rules can be found in the Coq development [Sammler et al. 2019].

Now we can generalize the proof rule for `syscall` to its final form, which allows us to use the system call assumptions A.[13]

P-SYSCALL-FULL

$$\frac{P(s, q_a, \Phi) \triangleq \begin{array}{l} \forall q_r\ st\ \pi.\ I_{SP}\ st \mathrel{-\!\!*} \\ \quad \exists \bar{i}.\ (\forall i \in \bar{i}.\ \exists o.\ \mathsf{in\_trace}\ i\ o) * \\ \quad\quad \forall\ i_{new} > max(\bar{i}).\ \mathsf{in\_trace}\ i_{new}\ (s, q_a, q_r) \mathrel{-\!\!*} \mathsf{full\_trace}\ \pi \mathrel{-\!\!*} A\ \pi \Rrightarrow \\ \quad\quad\quad \exists st'.\ st \overset{(s, q_a, q_r)\checkmark}{\longrightarrow}_{SP} st' * I_{SP}\ st' * \mathsf{full\_trace}\ \pi * \Phi\ q_r \end{array}}{\{P(s, q_a, \Phi)\}\ \textbf{syscall}(s, q_a)\ \textbf{at High}\ \{q_r\ \textbf{at High}.\Phi\ q_r\}_?}$$

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 $\bar{i}$ which have occurred in the trace ($\forall i \in \bar{i}.\ \exists o.\ \mathsf{in\_trace}\ i\ o$). In the proof for T-FILE-OPEN this is instantiated with $\bar{i}$ from I_{fileSP} .

Afterwards one receives an index i_{new} which is greater than all indices in $\bar{i}$ (which implies $i_{new} \notin \bar{i}$), and one knows that this index is now in the trace with the data of the current system call ($\mathsf{in\_trace}\ i_{new}\ (s, q_a, q_r)$). Furthermore, one gets the $\mathsf{full\_trace}\ \pi$ predicate and learns that the system call assumptions A hold for the current trace π. The rest is the same as P-SYSCALL except that one has to return ownership of $\mathsf{full\_trace}\ \pi$ 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 $\mathsf{file\_token}\ z_f$—which is used to establish the $[\![\textbf{file}]\!]$ corresponding to the **file** returned by T-FILE-OPEN—and the $\mathsf{in\_trace}\ i\ (s, q_a, q_r)$ 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

[13]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 $\Pi(\overline{\tau})$.

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 $\mathbf{Sealed}_{s,\tau}$ 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 λ_{sandbox}

This appendix contains the complete operational semantics of λ_{sandbox}.

$$h \in \mathit{Heap} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \qquad o \in \mathit{Observation} \triangleq (\mathit{Name}, \mathit{Lit}, \mathit{Lit}) \qquad \pi \in \mathit{Trace} \triangleq \mathit{list}\ \mathit{Observation}$$

$$\mathcal{I} \in \mathit{Interface} \triangleq (\mathit{Priv}, \mathit{Name}) \xrightarrow{\text{fin}} \mathit{Val} \qquad A \in \mathit{SyscallAssumptions} \triangleq \mathit{Trace} \to \mathit{Prop}$$

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

$$\boxed{h \vdash q_1 = q_2, h \vdash q_1 \neq q_2}$$

$$h \vdash l = l \qquad h \vdash z = z \qquad \frac{l_1 \notin \mathit{dom}(h) \lor l_2 \notin \mathit{dom}(h)}{h \vdash l_1 = l_2} \qquad \frac{z_1 \neq z_2}{h \vdash z_1 \neq z_2} \qquad \frac{l_1 \neq l_2}{h \vdash l_1 \neq l_2}$$

$$\boxed{p_1 \sqsubseteq p_2}$$

$$p \sqsubseteq p \qquad \textbf{Low} \sqsubseteq \textbf{High}$$

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.

$$\boxed{\mathcal{I} \vdash h, \pi \mid e \,\textbf{at}\, p \to h', \pi' \mid e_1' \,\textbf{at}\, p_1', (e_2' \,\textbf{at}\, p_2')^?}$$

O-ADD
$$\frac{z_1 + z_2 = z'}{\mathcal{I} \vdash h, \pi \mid z_1 + z_2 \,\textbf{at}\, p \to_A h, \pi \mid z' \,\textbf{at}\, p}$$

O-SUB
$$\frac{z_1 - z_2 = z'}{\mathcal{I} \vdash h, \pi \mid z_1 - z_2 \,\textbf{at}\, p \to_A h, \pi \mid z' \,\textbf{at}\, p}$$

O-LE-TRUE
$$\frac{z_1 \leq z_2}{\mathcal{I} \vdash h, \pi \mid z_1 \leq z_2 \,\textbf{at}\, p \to_A h, \pi \mid 1 \,\textbf{at}\, p}$$

O-LE-FALSE
$$\frac{z_1 > z_2}{\mathcal{I} \vdash h, \pi \mid z_1 \leq z_2 \,\textbf{at}\, p \to_A h, \pi \mid 0 \,\textbf{at}\, p}$$

O-EQ-TRUE
$$\frac{h \vdash q_1 = q_2}{\mathcal{I} \vdash h, \pi \mid q_1 = q_2 \,\textbf{at}\, p \to_A h, \pi \mid 1 \,\textbf{at}\, p}$$

O-EQ-FALSE
$$\frac{h \vdash q_1 \neq q_2}{\mathcal{I} \vdash h, \pi \mid q_1 = q_2 \,\textbf{at}\, p \to_A h, \pi \mid 0 \,\textbf{at}\, p}$$

O-OFFSET
$$\frac{l' = l + z}{\mathcal{I} \vdash h, \pi \mid l.z \,\textbf{at}\, p \to_A h, \pi \mid l' \,\textbf{at}\, p}$$

O-APP
$$\mathcal{I} \vdash h, \pi \mid (\texttt{rec}\, f\ (\overline{x}) = e)(\overline{v}) \,\textbf{at}\, p \to_A h, \pi \mid e[\texttt{rec}\, f\ (\overline{x}) = e/f, \overline{v}/\overline{x}] \,\textbf{at}\, p$$

O-READ
$$\frac{h(l) = v}{\mathcal{I} \vdash h, \pi \mid !l \,\textbf{at}\, p \to_A h, \pi \mid v \,\textbf{at}\, p}$$

O-WRITE
$$\frac{h(l) = v' \qquad l = (p', z_b, z_o) \qquad p' \sqsubseteq p}{\mathcal{I} \vdash h, \pi \mid l \leftarrow v \,\textbf{at}\, p \to_A h[l \leftarrow v], \pi \mid \text{☣} \,\textbf{at}\, p}$$

O-ALLOC
$$\frac{z > 0 \qquad l = (p', z_b, z_o) \qquad \{(p', z_b)\} \times \mathbb{Z} \# \mathit{dom}(h) \qquad p' \sqsubseteq p}{\mathcal{I} \vdash h, \pi \mid \texttt{alloc}_{p'}(z) \,\textbf{at}\, p \to_A h[l + m \leftarrow \text{☣} \mid 0 \leq m < z], \pi \mid l \,\textbf{at}\, p}$$

O-FREE

$$\frac{\begin{array}{c} z > 0 \\ l = (p', z_b, z_o) \qquad dom(h) \cap \{(p', z_b)\} \times \mathbb{Z} = \{(p', z_b)\} \times \{m | z_o \le m < z_o + z\} \qquad p' \sqsubseteq p \end{array}}{\mathcal{I} \vdash h, \pi \mid \texttt{free}(z, l)\ \textbf{at}\ p \longrightarrow_A h[l + m \leftarrow \bot \mid 0 \le m < z], \pi \mid \text{☣}\ \textbf{at}\ p}$$

O-CAS-FAIL

$$\frac{h(l) = q' \qquad h \vdash q' \neq q \qquad l = (p', z_b, z_o) \qquad p' \sqsubseteq p}{\mathcal{I} \vdash h, \pi \mid \texttt{CAS}(l, q, v)\ \textbf{at}\ p \longrightarrow_A h, \pi \mid 0\ \textbf{at}\ p}$$

O-CAS-SUC

$$\frac{h(l) = q' \qquad h \vdash q' = q \qquad l = (p', z_b, z_o) \qquad p' \sqsubseteq p}{\mathcal{I} \vdash h, \pi \mid \texttt{CAS}(l, q, v)\ \textbf{at}\ p \longrightarrow_A h[l \leftarrow v], \pi \mid 1\ \textbf{at}\ p}$$

O-CASE

$$\mathcal{I} \vdash h, \pi \mid \texttt{case}\ z\ \texttt{of}\ \overline{e}\ \textbf{at}\ p \longrightarrow_A h, \pi \mid e_z\ \textbf{at}\ p$$

O-FORK

$$\mathcal{I} \vdash h, \pi \mid \texttt{fork}\ e\ \textbf{at}\ p \longrightarrow_A h, \pi \mid \text{☣}\ \textbf{at}\ p, e\ \textbf{at}\ p$$

O-SYSCALL

$$\frac{\textbf{High} \sqsubseteq p \qquad A\,(\pi +\!\!+ [(s, q_a, q_r)])}{\mathcal{I} \vdash h, \pi \mid \texttt{syscall}(s, q_a)\ \textbf{at}\ p \longrightarrow_A h, \pi +\!\!+ [(s, q_a, q_r)] \mid q_r\ \textbf{at}\ p}$$

O-GATED-CALL

$$\frac{p = \textbf{High} \Leftrightarrow p' = \textbf{Low} \qquad \mathcal{I}((p', s)) = v}{\mathcal{I} \vdash h, \pi \mid \texttt{gated\_call}(s, \overline{v})\ \textbf{at}\ p \longrightarrow_A h, \pi \mid \texttt{gated\_ret}(v\ (\overline{v}))\ \textbf{at}\ p'}$$

O-GATED-RET

$$\frac{p = \textbf{High} \Leftrightarrow p' = \textbf{Low}}{\mathcal{I} \vdash h, \pi \mid \texttt{gated\_ret}(v)\ \textbf{at}\ p \longrightarrow_A h, \pi \mid v\ \textbf{at}\ p'}$$

O-MK-LOCATION

$$\frac{p' = \textbf{Low} \Leftrightarrow z_p = 0 \qquad z_b \ge 0 \qquad l = (p', z_b, z_o)}{\mathcal{I} \vdash h, \pi \mid \texttt{mk\_location}(z_p, z_b, z_o)\ \textbf{at}\ p \longrightarrow_A h, \pi \mid l\ \textbf{at}\ p}$$

O-GET-PRIV

$$\frac{l = (p', z_b, z_o) \qquad p' = \textbf{Low} \Rightarrow z_p = 0 \qquad p' = \textbf{High} \Rightarrow z_p = 1}{\mathcal{I} \vdash h, \pi \mid \texttt{get\_priv}(l)\ \textbf{at}\ p \longrightarrow_A h, \pi \mid z_p\ \textbf{at}\ p}$$

O-GET-BLOCK

$$\frac{l = (p', z_b, z_o)}{\mathcal{I} \vdash h, \pi \mid \texttt{get\_block}(l)\ \textbf{at}\ p \longrightarrow_A h, \pi \mid z_b\ \textbf{at}\ p}$$

O-GET-OFFSET

$$\frac{l = (p', z_b, z_o)}{\mathcal{I} \vdash h, \pi \mid \texttt{get\_offset}(l)\ \textbf{at}\ p \longrightarrow_A h, \pi \mid z_o\ \textbf{at}\ p}$$

$$\boxed{\mathcal{I} \vdash h, \pi \mid E \longrightarrow_A h', \pi' \mid E'}$$

O-Thread

$$\frac{\mathcal{I} \vdash h, \pi \mid e \,\mathbf{at}\, p \rightarrow_A h', \pi' \mid e'_1 \,\mathbf{at}\, p'_1, (e'_2 \,\mathbf{at}\, p'_2)^?}{\mathcal{I} \vdash h, \pi \mid E \mathbin{+\!\!+} [K[e] \,\mathbf{at}\, p] \mathbin{+\!\!+} E' \longrightarrow_A h', \pi' \mid E \mathbin{+\!\!+} [K[e'_1] \,\mathbf{at}\, p'_1] \mathbin{+\!\!+} E' \mathbin{+\!\!+} [e'_2 \,\mathbf{at}\, p'_2]}$$

ACKNOWLEDGMENTS

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

Martín Abadi. 1997. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings (Lecture Notes in Computer Science)*, Martín Abadi and Takayasu Ito (Eds.), Vol. 1281. Springer, 611–638. https://doi.org/10.1007/BFb0014571

Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When good components go bad: Formally secure compilation despite dynamic compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 1351–1368. https://doi.org/10.1145/3243734.3243745

R.P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. 1976. *Security analysis and enhancements of computer operating systems.* Technical Report. Institute for Computer Sciences and Technology, National Bureau of Standards, Washington,D.C. https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nbsir76-1041.pdf

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.* 32, 3 (2010), 7:1–7:67. https://doi.org/10.1145/1709093.1709094

ARM Limited. 2009. *ARM Security Technology.* Technical Report. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

David A. Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zalinescu. 2013. Enforceable security policies revisited. *ACM Trans. Inf. Syst. Secur.* 16, 1 (2013), 3:1–3:26. https://doi.org/10.1145/2487222.2487225

Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas P. Jensen, and Pierre Wilke. 2019. Compiling sandboxes: Formally verified software fault isolation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings.* 499–524. https://doi.org/10.1007/978-3-030-17184-1_18

Matthias Blume and David A. McAllester. 2004. A sound (and complete) model of contracts. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, Chris Okasaki and Kathleen Fisher (Eds.). ACM, 189–200. https://doi.org/10.1145/1016850.1016876

Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. 2011. Resource-aware authorization policies for statically typed cryptographic protocols. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011.* IEEE Computer Society, 83–98. https://doi.org/10.1109/CSF.2011.13

Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* IEEE Computer Society, 813–830. https://doi.org/10.1109/SP.2015.55

Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce. 2018. The meaning of memory safety. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Lujo Bauer and Ralf Küsters (Eds.), Vol. 10804. Springer, 79–105. https://doi.org/10.1007/978-3-319-89722-6_4

Akram El-Korashy. 2016. *A Formal Model for Capability Machines: An Illustrative Case Study towards Secure Compilation to CHERI.* Master's thesis. Universität des Saarlandes. http://hdl.handle.net/11858/00-001M-0000-002C-41CA-B

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 48–59. https://doi.org/10.1145/581478.581484

Cormac Flanagan. 2006. Hybrid type checking. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 245–256. https://doi.org/10.1145/1111037.1111059

Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2007. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 25. https://doi.org/10.1145/1275497.1275500

Timothy S. Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 268–277. https://doi.org/10.1145/113445.113468

Google. 2019. Sandboxed Api. https://github.com/google/sandboxed-api.

Andrew D. Gordon and Alan Jeffrey. 2001. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada.* 145–159. https://doi.org/10.1109/CSFW.2001.930143

Orna Grumberg and David E. Long. 1994. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 843–871. https://doi.org/10.1145/177492.177725

Norman Hardy. 1988. The confused deputy (or why capabilities might have been invented). *Operating Systems Review* 22, 4 (1988), 36–38. https://doi.org/10.1145/54289.871709

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On polymorphic gradual typing. *PACMPL* 1, ICFP (2017), 40:1–40:29. https://doi.org/10.1145/3110284

Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. 2016. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016.* 45–60. https://doi.org/10.1109/CSF.2016.11

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. Rustbelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 437–452. https://doi.org/10.1145/3064176.3064217

Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic sandboxing of unsafe components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems, Shanghai, China, October 28, 2017*, Julia Lawall (Ed.). ACM, 51–57. https://doi.org/10.1145/3144555.3144562

Nico Lehmann and Éric Tanter. 2017. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 775–788. http://dl.acm.org/citation.cfm?id=3009856

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2.* Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, Ruby B. Lee and Weidong Shi (Eds.). ACM, 10. https://doi.org/10.1145/2487726.2488368

James H. Morris, Jr. 1973. Protection in programming languages. *Commun. ACM* 16, 1 (1973), 15–21. https://doi.org/10.1145/361932.361937

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. Rocksalt: Better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012.* 395–404. https://doi.org/10.1145/2254064.2254111

Mozilla. 2019. *Script security.* Technical Report. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Script_security

Max S. New and Amal Ahmed. 2018. Graduality from embedding-projection pairs. *PACMPL* 2, ICFP (2018), 73:1–73:30. https://doi.org/10.1145/3236768

Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2019. The high-level benefits of low-level sandboxing – Artifact. https://doi.org/10.5281/zenodo.3533037 Latest version: https://gitlab.mpi-sws.org/FCS/lang-sandbox-coq.

Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50. https://doi.org/10.1145/353323.353382

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *PACMPL* 1, OOPSLA (2017), 89:1–89:26. https://doi.org/10.1145/3133913

The Rust Developers. 2019. The Rust programming language. https://rust-lang.org.

Jesse A. Tov and Riccardo Pucella. 2010. Stateful contracts for affine types. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Andrew D. Gordon (Ed.), Vol. 6012. Springer, 550–569. https://doi.org/10.1007/978-3-642-11957-6_29

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2019. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.* 3, ICFP, Article 84 (July 2019), 29 pages. https://doi.org/10.1145/3341688

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 249–257. https://doi.org/10.1145/277650.277732

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society, 79–93. https://doi.org/10.1109/SP.2009.25