Master Thesis

Automatic Sandboxing of Unsafe Software Components in High Level Languages

Benjamin Lamowski

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dr. Carsten Weinhold

3. Mai 2017

Aufgabenstellung

Neue “sichere“ Programmiersprachen wie Go, Swift oder Rust wurden nicht nur für die normale Anwendungsentwicklung entworfen, sondern sie zielen auch auf eine hochperformante Ausführung und Programmierung vergleichsweise systemnaher Funktionalität ab. Eine attraktive Eigenschaft beispielsweise von Rust ist das gegenüber C und C++ deutlich strengere Speicherverwaltungsmodell, bei dem bereits zur Kompilierzeit der Lebenszyklus und die Erreichbarkeit von Objekten sowie die Zuständigkeit für deren Allokation und Deallokation wohldefiniert sind. Ganze Klassen von Programmfehlern wie etwa Buffer Overflows oder Dereferenzierung ungültige Zeiger werden dadurch eliminiert und die Programme mithin sicherer und robuster.

Aus diversen Gründen müssen Programme, die in sicheren Sprachen geschriebenen wurden, aber oftmals auf “unsicheren“ Legacy-Code zurückgreifen. So bietet etwa Rust über das “unsafe“-Sprachelement die Möglichkeit, Funktionen innerhalb von Bibliotheken aufzurufen, die in fehleranfälligem C geschrieben sind. Leider werden die vom Compiler durchgesetzten Garantien der sicheren Sprache hinfällig, sobald im Code einer C-Bibliothek ein Speicherfehler auftritt. Ein Schreibzugriff etwa durch einen ungültigen Zeiger in der C-Bibliothek kann potenziell auch im selben Adressraum befindlichen Programmzustand modifizieren, der von Programmteilen in der sicheren Sprache und deren Laufzeitsystem verwaltet wird.

Ein vielversprechender und am Lehrstuhl Betriebssysteme schon mehrfach erfolgreich demonstrierter Ansatz zur Isolation “sicherer“ und “unsicherer“ Teile eines Softwaresystems besteht darin, die Programmfunktionalität und Datenhaltung auf mehrere Adressräume zu verteilen.

Im Rahmen der Master-Arbeit soll untersucht werden, wie ein derartiger Adressraumschutz zwischen Code und Daten des in der sicheren Sprache geschriebenen Programmteils und dem potenziell unsicheren Legacy-Code umgesetzt werden kann. Dabei soll eine möglichst nahtlose Integration und Anlehnung an Konstrukte der sicheren Programmiersprache angestrebt werden. Hierzu soll das Laufzeitsystem und gegebenenfalls der Compiler entsprechend modifiziert und erweitert werden. Ziel ist es, dass Legacy-Code weitgehend automatisch und transparent in einem separaten Adressraum ausgeführt wird und der Datenaustausch zwischen den Programmteilen möglichst wenig Aufwand für den Programmierer erfordert.

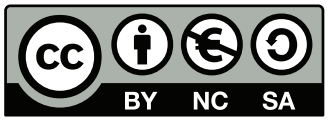
Die Lösung soll für eine geeignete Programmiersprache prototypisch umgesetzt und anhand mindestens einer Beispielanwendung bewertet werden. Dabei sind insbesondere die Performance sowie der Zugewinn and Robustheit und Sicherheit zu betrachten. Im Rahmen der Arbeit kann davon ausgegangen werden, dass ein geeigneter “Sandboxing“-Mechanismus (z.B. SELinux oder Systemaufruf-Filter) vorhanden ist. Die genauen Anforderungen an diesen sollen aber analysiert und in der Arbeit diskutiert werden.

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 3. Mai 2017

Benjamin Lamowski



Abstract

This work explores the design space of automated componentization for software written in modern safe programming languages, with the goal of sandboxing unsafe legacy libraries. It describes the design and implementation of *Sandcrust*, a Rust library that enables the reuse of C libraries in Rust without compromising the memory safety guarantees for safe Rust code. The Linux prototype shows that it is possible to safely use complex C library interfaces, while providing seamless integration into the Rust development ecosystem. The performance evaluation identifies a performance bottleneck in Bincode, a popular Rust library, and quantifies the impact by implementing a common use case without Bincode. Alternatives for abstracting a paradigm of componentization in a programming language are examined for the use case of separating an untrusted external component.

Contents

List of Illustrations

List of Tables

List of Figures

List of Listings

1 Introduction

The reason we all like to think so well of others is that we're all afraid for ourselves. The basis of optimism is sheer terror.
— Oscar Wilde

Traditional systems programming languages like C expose attack vectors that threaten the safety of the program. These vulnerabilities stem from a lack of memory safety enforcement that may lead to unintended control flows and crashes to the point of malicious code execution. This notwithstanding, C is still one of the most widely used programming languages to date [1, 2]. Modern programming languages like Rust[1] provide safe primitives and enforce strict rules for memory access and error handling at compile time, effectively limiting the causes of unsafe behavior to bugs in the compiler and runtime implementation. While this does not rule out the possibility of safety-critical program failures, it guarantees the controlled execution of arbitrary software written in safe Rust, if no safety-critical faults are present in the language implementation.

Because of their relative novelty, these languages often support reusing existing software, especially by offering bindings to C-language libraries. However, because virtually all commodity operating systems (OSs) use processes as the primary protection domain, any vulnerability in a legacy library will affect the whole process, foiling the guarantees provided by the new language.

Language-specific package managers provide easy integration of third party code. Rust puts a huge emphasis on memory safety, but many third party libraries (*crates*) do in fact use libraries written in C[2]. Rust's crate system provides excellent modularization at the source level, but this separation is not retained, as it is subject to the restrictions expressed by Strackx and Piessens: „High-level programming languages offer protection facilities such as abstract data types, private field modifiers, or module systems. While these constructs were mainly designed to enforce software engineering principles, they can also be used as building blocks to provide security properties. Declaring a variable holding a cryptographic key as private, for example, prevents direct access from other classes. This protection however does not usually remain after the source code is compiled. An attacker with in-process or kernel level access is not bound by the type system of the higher language“ [3, p. 3].

The rich tradition of privilege separation, especially in μkernel-based operating systems, has as of yet failed to be reflected in application development. Modern languages like Rust and Go[3] provide first class abstractions for concurrency. But the lack of stringent access

[1] `https://www.rust-lang.org/`
[2] As an illustration: as of this writing, crates.io lists 1209 crates depending on the *libc* crate: `https://crates.io/crates/libc/reverse_dependencies`
[3] `https://golang.org/`

restrictions to OS resources leads to an enduring dominance of monolithic applications and the absence of advanced language abstractions for privilege separation.

In the face of widespread vulnerabilities, some high profile applications like web browsers have employed *sandboxing* to contain especially vulnerable software components. This mimics composed multimedia applications as found in L4 μkernel based DROPS [4] by manually setting up a restricted environment for processes by the unrestricted main application.

Safe languages like Rust are gradually being used to replace unsafe components like the MP4 decoder in the Firefox browser[4] and there is a project to automatically convert C code to Rust[5]. Rust is being ported to μkernel OS's like the Genode OS Framework[6] and seL4[7], and even used to build a new μkernel OS[8]. Still, as Jamey Sharp has remarked, the pressure to ensure not merely safety but correctness may make the reuse of an existing component preferable over a rewrite of that component in a safe language [5].

A vast body of research has explored the disaggregation of monolithic software, but widespread adoption of these techniques is impeded by the amount of manual conversion work, wanting integration in existing development toolchains, and reliance on custom OS extensions. Recent operating systems include abstractions that cater to the demand for sandboxing. The effect of modern programming languages and toolchains on software disaggregation has gone unexamined until now.

The key motivation for this work is that modern high level languages open up a new design space for automated sandboxing of unsafe software components. This design space is explored with a focus on integration with the existing Rust ecosystem and language. Previous efforts to contain Rust components with unsafe code have been deprecated upstream[9]. The goal of this work is to fill this gap, providing easy to use isolation of untrusted legacy components by executing them in a sandbox. This enables safe Rust programs to use of existing software written in unsafe languages like C without invalidating Rust's memory safety guarantees. With the *Sandcrust* (**Sand**boxing **C** in **Rust**) isolation crate, a prototype implementation is provided and its security properties and limitations for use with real-world legacy libraries are evaluated.

The remainder of this work is organized as follows: The next chapter provides a background on theoretical concepts, related work and the implementation environment. Chapter 3 discusses the assumptions and design of the prototype. The implementation is detailed in Chapter 4 and the prototype evaluated in Chapter 5. Chapter 6 concludes and gives an outlook on possible future directions.

[4] `https://hacks.mozilla.org/2016/07/shipping-rust-in-firefox/`
[5] `https://github.com/jameysharp/corrode`
[6] `https://genode.org/documentation/release-notes/16.05`
[7] `https://robigalia.org/`
[8] `https://www.redox-os.org/`
[9] `https://internals.rust-lang.org/t/safe-rust-sandboxing-untrusted-modules-at-api-level/1505`

2 Technical Background

To the uneducated, an A is just three sticks.
— Winnie the Pooh

This chapter introduces key theoretical concepts in Section 2.1, discusses related work in Section 2.2, and gives an overview on important aspects of the implementation environment in Section 2.3.

2.1 Theoretical Concepts

After a differentiation of *unsafe software components*, this section will turn to a solution of the problem and introduce the principles and practice of *privilege separation*.

2.1.1 Unsafe Software Components

As a starting point, we form a concept of *safety* following Avizienis et al., who define it as „absence of catastrophic consequences on the user(s) and the environment“ [6, p. 13]. But what leads software to cause catastrophic consequences? An undiscerning end user might mistakenly execute *malware*, whose sole function serves an attacker, causing catastrophic consequences for the user. A component, however, is included in software, because it can establish desired functionality. Unsafe behavior is a severe form of *failure*: non-compliance of a system to its agreed function. Usually unintended, this failure is caused by a *fault*, or bug if a programming mistake, that has led to an *error* in the state of the system.

The naïve response to that is a call for pure, bug free software, but it is not that easy.

Fault metrics. Various metrics have been proposed to quantify the number of faults, and by extension the safety of software. In terms of quantity, the notion that the amount of faults is correlated to the Source Lines of Code (SLOC) [7, 8], seems to date back to work by Akiyama in 1971 [9]. It is countered by Ferdinand's detailed mathematical analysis in [10] that concludes that even though componentization may increase the absolute size of the program, it helps reduce the number of programing errors. This conclusion, however, was criticized along with many other metrics by Fenton and Neil [11]. They propose Bayesian Belief Networks (BBNs) as a solution to the problem [12], while Nagappan, Ball, and Zeller propose a combination of traditional metrics [13].

Whatever the specific metric, the authors agree in that there is a correlation between some measure for complexity and the number of faults. Some software authors conclude

that "small" software is the solution to the problem[1], but while badly written software obviously is more prone to errors, their purist approach simply cuts out the features or convenience that are required of more complex software, and their software is therefore often limited in use to a circle of enthusiasts.

Language safety. When a software component is used for the complexity of the problem it solves, safety-critical errors need to be eliminated by limiting the consequences of errors to safe program states, or disallowing program code that may exhibit unsafe behavior.

Unfortunately, more traditional languages, especially C and by extension C++, offer little towards that path, contrary to modern languages like Rust, as will be explored in detail in Section 2.3.1.

When unsafe software does not generally exhibit unsafe behavior, it must be triggered directly or indirectly by program input. In C, writes to memory are not validated by design, which may allow input to directly overwrite memory. Other problems include the lack of memory pointer sanitation, or an integrated error handling mechanism. These design omissions can often be exploited for code injection [14, 15, 16, 17, 18], while advanced countermeasures on the system level like address space layout randomization (ASLR) [19, 20] or memory that is exclusively writeable *or* executable [21] are not always effective. Even compiler authors do often not agree on the semantics of the C language [22].

When many of today's available software libraries are written in unsafe languages, there needs to be a way to limit the scope of unsafe behavior.

2.1.2 Privilege Separation

Privilege separation is the theory and practice to limit the power of a software component to the privileges necessary to serve its purpose. Before getting an overview of the practical work in the field in the next section, this section introduces the underlying theory.

The Principle of Least Privilege. In their classic paper *The Protection of Information in Computer Systems* from 1975, Saltzer and Schroeder describe the Principle of Least Privilege as follows:

> Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. [23, p. 1282]

Componentization. Componentization is the disaggregation of software into functional components according to this principle. Besides providing the base for privilege separation, componentization may enable concurrency in execution or development through a new software design, at the expense of additional component interface complexity.

[1] Prominent proponents include Daniel J. Bernstein (`https://cr.yp.to/djb.html`), Felix von Leitner (`https://www.fefe.de/dietlibc/diet.pdf`) and the contributers to suckless.org projects (`https://suckless.org/philosophy`)

Trusted Computing Base. A componentized system relies on the Trusted Computing Base (TCB) to ensure that least privilege of components is enforced. Initially defined by Rushby as „the combination of kernel and trusted processes“ [24, p. 13], the expanded definition in The US Department of Defense's "Trusted Computer Systems Evaluation Criteria“ [25, p. 66] gives a better idea of it's range:

> The heart of a trusted computer system is the Trusted Computing Base (TCB) which contains all of the elements of the system responsible for supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based. [...] a TCB should be as simple as possible consistent with the functions it has to perform. Thus, the TCB includes hardware, firmware, and software critical to protection and must be designed and implemented such that system elements excluded from it need not be trusted to maintain protection.

The strong consistency demanded in the TCB definition above is not provided by general purpose systems. Instead, they limit themselves to restricting particularly untrusted components.

Sandboxing. With the focus on unsafe software components in prevailing monolithic software designs, such a restriction is called a sandbox. Initially used by Wahbe et al. to describe isolation that is only software-defined [26], Goldberg et al. defined the term to mean a combination of OS application programming interface (API) restrictions supported by hardware isolation [27].

2.2 Related Work

Starting with systems that are designed for privilege separation, this section moves on to efforts to contain software in a sandbox, before providing a classification of efforts to componentize existing software.

2.2.1 Privilege Separation by Design

Systems designed with a focus on privilege separation form a standard by which to judge attempts on componentizing monolithic software, so a brief overview over work in the field is in order.

Capabilities, introduced by Dennis and Van Horn in [28], are unforgeable tokens to a resource. Where Saltzer and Schroeder's 1975 exploration of OS protection [29] still describes a mixed use with Access Control Lists (ACLs), capabilities quickly became the dominant concept for componentized systems.

Multics made use of hardware capabilities, but was eventually dominated by Unix, limited to the "one job, one tool“ philosophy [30] and simple ACLs, when computers lacking the hardware capabilities envisioned in pure capability systems like PSOS [31] and lately revived in CHERI [32], gained widespread use. Microkernel operating systems continued the tradition and implemented object capabilities in software: starting with

Hydra [33]; Mach [34], L4 [35, 36], EROS [37] and many others explore the design space of maximum privilege separation and minimal TCB.

Meanwhile, componentized software for commodity OS's like *qmail*[2] or *Postfix*[3] had to employ a combination of different user and group IDs and ACL protected data to achieve any privilege separation. Recently, OS vendors support compartmentalization as part of the API, like Apple's *XPC*[4].

Several solutions have been proposed that avoid traditional process-based protection domains altogether: Banerji, Tracey, and Cohn propose services dynamically mapped as libraries via segments as a way to implement services while preserving protection boundaries [38], similar to the *overlapping* scheme in Opal [39]. The Singularity research OS uses Software-Isolated Processess (SIPs) with a *sealed process architecture* [40] albeit also supporting hardware isolation [41].

This is in concept similar to some forms of sandboxing which is the subject of the next section.

2.2.2 Sandboxing

The lack of privilege separation in the design of commodity software has been a growing security concern in the face of increasingly complex software, like web browsers as an ubiquitous application platform. Consequently, the initial work in this field contains helper applications in the Netscape browser by restricting their access to system calls, a technique refined in later work [27]. Where Jensen and Hagimont and various others implement a reference monitor in user space using ACLs [42] and Jain and Sekar use process tracing facilities [43], later work such as FreeBSD Jails [44], Systrace [45] and Seccomp-BPF [46] has been implemented as extensions to the monolithic kernel.

The most prevalent work to date is SELinux [47]. It has gained notoriety for its complex, defensive security policies and serves as a prefect illustration of the difficulties of fencing in a broad monolithic interface, compared to the powerful primitives found in μkernel OS's.

Explicitly „eschewing microkernel design“ [48, p. 98], Capsicum extends the Unix API with capabilities to provide generalized sandboxing from within the application, while Shinagawa, Kono, and Masuda blend user-level reference monitors with system call interception to allow for flexible sandboxing strategies [49].

The *Android Application Sandbox* [50] is likely the most widely used sandboxing mechanism in consumer software. It is conceptionally similar to early separation in web browsers, which are in the process of migrating to more modern facilities such as Seccomp-BPF in combination with additional abstractions we will describe in Section 2.3.2 [51].

On the server side of distributed applications, *Docker*[5] and *Rkt*[6] use the same sandboxing mechanism for containerization. *Flatpak*[7] (formerly *xdg-app*) mirrors this approach

[2] `http://cr.yp.to/qmail.html`
[3] `https://postfix.org/`
[4] `https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingXPCServices.html`
[5] `https://www.docker.com/`
[6] `https://coreos.com/rkt/`
[7] `http://flatpak.org/`

for desktop applications. External sandboxing of individual applications is provided by *Mbox* [52], Firejail[8] and *systemd-nspawn*[9]. They all have in common that the sandboxes are employed for *inter-application* isolation.

A second tradition of work moves closer to the idea of componentization within a program and uses Software Fault Isolation (SFI) to check isolation properties at run time: Building on early work by Wahbe et al. [26] and subsequent extensions [53, 54, 55], XFI [56] rewrites Microsoft Windows binaries to honor Control Flow Integrity (CFI). Returning to the theme of web browser extensions, the sandboxing of Native Client (NaCl) [57] applications rests on SFI prior to execution in a runtime with additional checks and has since been simplified in Rocksalt [58].

The state of system security has yielded some less obvious implementations of sandboxing. Nooks [59] is an in-kernel reference monitor to contain non-malicious faults in Linux device drivers. The browser built on top of SubOS [60, 61] attaches a *sub-user id* to downloaded files that are injected into the applications which open them, aiming to restrict the damage caused by malicious input. Virtics [62] shirks off Portable Data Format (PDF) rendering in the browser all the way to a virtual machine in an attempt to limit PDF exploits.

Sandbox design trade-offs are discussed in more detail in [63] and a comprehensive discussion of sandboxing is found in [64]. As the last examples show, sandboxing is a valuable tool to enforce privilege separation in a legacy OS, but does not itself assist in the process of the componentization of existing software, which is the last area and most related work we will look into.

2.2.3 Componentization of Existing Software

Work spanning more than four decades has examined the componentization of existing software from many different angles.

Protection Models for Componentization

A number of protection models have been identified, that in the following will be used as a classification:

- **Distribute across the network:** Distribute components across the network for improved availability or reliability.
- **Protect sensitive information:** Componentize software according to access to information.
- **Minimize privileged components:** Separate a high privilege part and run the rest of the software with lower privileges.
- **Reduce the TCB:** Reduce the TCB of a critical software component.
- **Restrict untrusted components:** Restrict the privileges of an untrusted software component.

Table 2.1 gives an overview over the work available in each field. The results of one model certainly have effects towards the others, but they still serve as a useful categorization.

[8] `https://firejail.wordpress.com/`
[9] `http://0pointer.de/public/systemd-man/systemd-nspawn.html`

For example, an effort focused on minimizing a program's privileged component will result in the reduction of the TCB for other programs that the privileges extend to, but it does not reduce the TCB of the componentized program: the minimal privileged component retains its power, and components within the unprivileged rest of the program still hold the same privileges over each other.

The remainder of this section will cover existing work in each field.

Distribute Across the Network

Perhaps surprisingly, the first published work in the field has a focus on distributing software across a computer network. ICOPS' raison d'être is the growing availability of „micro computers in homes, offices, and schools, both to provide programmable intelligence to terminals and peripherals, and to serve as local general-purpose computers for modest processing tasks“ [65, p. 484]. This early work already uses advanced mechanisms: calls to annotated procedures are trapped and a runtime is called instead to perform a Remote Procedure Call (RPC). The challenges and limitations are also comparable to later work: global variables are simply not supported, and „if data structures (or pieces thereof) are passed between the mainframe and satellite, it must be done in such a way as to avoid any pointer chasing of the ICP run-time environment [...]“ [65, p. 491].

CAGES [66], published shortly thereafter, shares the general concept and improves on the limitations by offering somewhat advanced signaling and support for (annotated) global variables. In 1999, Coign [67], designed for Microsoft's Component Object Model (COM), is first to propose libraries as the boundary at which to componentize applications. JIF/Split [68] and Swift [69] are both network-focussed advancements of data-centric approaches that will be introduced next.

Altogether, especially the early work in this field already offers valuable insight into the challenges of breaking up data structures of formerly monolithic software. However, although strong privilege separation is a side effect of distributed systems, the research is not easily applicable to componentization in today's highly different ecosystems.

Protect Sensitive Information

In 1997, Myers and Liskov approached componentization from a different angle by attaching security labels to data to enforce properties like integrity or confidentiality [70], resulting in JFlow [71], an extension of the Java programming language. The concept was later iterated on with JIF [72], a dynamic version of JIF [73], the aforementioned JIF/Split and a replication framework built on top of it [74].

Passe [75] has an unusual focus in manually adopting the Django web framework to, trained with test cases, enforce privilege separation for web applications.

With Flume [76], Krohn et al. combine the concept of Decentralized Information Flow Control (DIFC) with an access control mechanism at the system call level via deflection to a reference monitor.

Bittau et al. describe the most radical data-centric approach to date. In Wedge [77], the *Crowbar* dynamic analyzer identifies necessary access privileges and the program is then split up into *sthreads* that operate on least privilege and invoke *Callgates* to

Method	manual	annotate functions	annotate data	split on syscalls	split on library boundary
distribute across the network		ICOPS (1974), CAGES (1975)	JIF-Split (2001), Swift (2007)		Coign (1999)
protect sensitive information	Passe (2014)		JFlow (1999), JIF (2000), Flume (2007), Wedge (2008), Arbiter (2013)[2]		
minimize privileged components	OpenSSH (2003), Privman (2003)	Privtrans (2004)		ProgramCutter (2013)	
reduce TCB	Nizza (2005)[1], Xen (2008)[1]	Coir (2010)[12], TrustVisor (2010)[1], Fides (2012)[12]	SeCage (2015)[12]	Proxos (2006)[1]	
restrict untrusted components	Chrome (2009), uPro (2012)[2]	Quarantine (2011), Fracture (2014)			Codejail (2012)

[1] use Virtual Machine Monitor (VMM) separation
[2] run in same virtual address space
(all others) use process boundaries

Table 2.1: Classification of componentization work

other sthreads to perform privileged operations. The application to common real-world applications illustrates the complexity of a radical, data-centric approach: „For example, enforcing a boundary between Apache/OpenSSL's worker and master sthreads required identifying 222 heap objects and 389 globals. Missing even one of these results in a protection violation and crash under Wedge's default-deny model“ [77, p. 317]. The complexity is justified with catching design errors, were static analysis would assign privileges that are not necessary during correct operation, but could for example be exploited to leak data.

Lastly, Arbiter [78] is notable for its radically different separation mechanism. Focussing on modern multi-threaded services, they adopt HiStar's [79] label model for a special shared *Arbiter Secure Memory Segment* kernel abstraction governed by a reference monitor to provide data-centric privilege separation while supporting policies that surpass the privileged vs. unprivileged model.

The value of this line of work lies in it's radical application of the principle of least privilege. Unfortunately, while unmatched in its analysis, the complexity of the data-centric protection model comes close to a complete reimplementation of the existing program[10], which is prohibitive when existing components are reused precisely to save implementation effort.

Minimize Privileged Components

Recent research has applied data-centric approaches to common server software, but it is predated by work focussed on minimizing their privileged components.

Provos, Friedl, and Honeyman described their manual split up of the OpenSSH remote access server in 2003 [81]. In the same year, Kilpatrick published Privman, a library aiding manual privilege minimization for Unix servers [82].

Privtrans [83] is the first attempt to automate this process by annotating privileged function calls and performing the split into a *monitor* and *slave* automatically, using partially automatically derived wrapper functions to perform RPCs.

ProgramCutter [84] has a more data-driven approach. After labeling the privileges of system calls, a program is represented as a graph of functions as nodes weighted by SLOC and privileges, and edges weighted for the amount of data transfered between them. The twofold goal is then to minimize both via dynamic analysis. However, for each data type, serialization needs to be implemented by hand.

The practical focus of this protection model also constitutes its weakness: while well-suited to common server software, the narrow focus on privileged operations fails to meet the needs of compartmentalization of arbitrary component interfaces.

Reduce the Trusted Computing Base

While the last protection model is aimed at taking privileges away from much of the program, therefore taking it out of the TCB, the orthogonal approach is to reduce the external TCB that a trusted part of the program has to rely on.

[10] This resembles Marx' work on capitalism [80] remarkably, one should hope there is a bigger chance of success.

The Nizza Architecture [85] manually splits out critical parts of legacy applications running virtualized on a μkernel OS into *trusted wrappers* or *AppCores*, reducing their TCB by orders of magnitude [86]. As Weinhold and Härtig show with their implementation of the *VPFS* trusted file system [87, 88], with a careful security concept, the underlying design can be leveraged to utilize the untrusted from the trusted part.

Murray, Milos, and Hand's work on manually reducing the TCB of Xen [89] may seem similar, but only serves the purpose of elevating its architecture to the standard set years earlier by L^{4}Linux [90].

Proxos [91] has an interesting take on TCB reduction by annotating system calls to be routed to a *Private OS* depending on their arguments. Bootstrapping the thus secured application is done via a newly introduced `pr_execve` system call, that behind the scenes spawns the appropriate VM and sets up a fake process as an interface in the Linux kernel.

Similar to the ideas laid out by Murray and Hand in [92], Coir [93] separates software on the library level using a VMM to switch to a different protection domain when a function in a library, ostensibly in the same logical address space, is used. Where they propose a special page fault handler for trapping into the VMM to retain binary compatibility, Coir uses function annotation to rewrite sensitive function calls to trap into the VMM on demand.

Conceptually a mix between the previously mentioned work, TrustVisor [94] feeds a manually created specification of sensitive functions into a special linker that will place those functions in pages where access causes a trap into the VMM, so the execute transparently in an environment secured by a *dynamic root of trust for measurement (DRTM)*.

Fides [3] has very similar objectives, but provides a more modern interface by extending LLVM[11] to support code annotation. A DRTM protected *vault* verifies *Self-Protecting Modules* in the same address space that are executed on a *security kernel*.

Latest in the field, SeCage [95] extends the general theme by combining static and dynamic analysis to identify secrets and using a trampoline to load new Extended Page Tables (EPTs) via `VMFUNC` without VMM intervention.

As this section has shown, the goal of TCB reduction necessitates a radically different runtime environment of the different components. This does not align well with the stated preference of seamless integration with existing development setups, but is an interesting direction for automatic separation to be explored in Conclusion and Outlook.

Restrict Untrusted Components

For the amount of error-prone libraries, exploration of the idea to restrict untrusted components of an application has started surprisingly late.

Again, the first work in this field was done manually, by splitting off a sandboxed rendering process from the Chrome web browser main program [96]. uPro [97] is also based on manual splitting decisions, but generates wrapper functions automatically and uses SFI to enforce separation within one address space with the uPro runtime.

[11] `https://www.llvm.org/`

Function level isolation boundaries are placed dynamically in Quarantine [98], depending on run time dependability, aided by manual conversion of C code to the *Quarantine Programming Model*. Its successor Fracture [99] is similar, but with comprehensive componentization, restart and replication options, optionally derived from the *Intelligent Boundary Subsystem*.

Codejail [100] is the most important precursor to this work. It restricts untrusted libraries by interpositioning a *trusted library* by hooking into `libc start main` and using `LD_PRELOAD`. Functions in the untrusted library are then run in a sandboxed second process. Its main advantage is that it does not require changes to the main program, nor to the untrusted library. However, this flexibility is paid dearly with the relatively large number of manual programing necessary to implement a wrapper library or modify the main program directly.

Especially Codejail deserves praise for their straightforward approach to a security issue that plagues so much common software. As the first comprehensive solution to the problem, it lacks integration into the development process, a focus of the design we will explore in the next chapter.

2.3 Implementation Environment

This section gives a short introduction to the implementation environment, specifically the Rust programming language and Linux sandboxing mechanisms.

2.3.1 The Rust Programming Language and Runtime

Rust was chosen as a programming language, because it provides strong memory safety guarantees as well as a *Foreign Function Interface (FFI)* to interface with existing libraries written in less safe programming languages like C.

Memory Safety Quarantees

Rust is statically typed [101] and its standard library types go to great lengths to ensure that memory errors like out-of-bounds accesses are not possible, thus eliminating whole classes of vulnerabilities common in C and C++, as described in Section 2.1.1.

Traits. In Rust, *traits* are often used to signal some form of functionality of a data type, similar to a class implementing an interface in object-oriented programming languages. Extending the concept, Rust has a number of built-in special traits that are used to provide generic behavior of a data type by implementing a function that is automatically called in certain situations. Additionally, structs as a composite of simple types can automatically *derive* traits if implemented for their member types.

Variable lifecycle. At first glance, Rust exhibits little difference to a modern procedural programming language like Go. This changes at the first try of re-assigning a variable, because variables are read-only by default. Rust uses Resource Acquisition Is Initialization (RAII). This is the foundation for Rust's memory safety, since it rules out uninitialized variables. The other end of a variable's life cycle is defined by *Lifetimes*,

which are calculated out of a binding's *scope* via *Lifetime Elision* for common cases or explicitly provided to resolve ambiguities; and which ensure that resources will be freed deterministically in the reverse order of their introduction, preventing use-after-free errors [102].

On a semantic level, the Rust standard library enforces explicit error handling by use of the `Option` and `Result` return types that require explicit matching on the result or error condition; or at least the call of the `unwrap()` method, which will panic and terminate the program in a controlled way in case of error.

Concurrency. As a cornerstone of concurrency, passing variables into a different scope, like most usually done by passing arguments to a function, *moves* the variable binding into that scope, depriving the original scope from access. This strict notion of *Ownership* [103] is necessary to extend the protection from use-after-free errors to concurrently running program routines. To allow for retaining a variable after use in a function call, *References* can be used to *borrow* the variable [104]. At this point, the explicit annotation mutable of variables comes in useful as it allows for compile-time detection of reader-writer race conditions. As references are still quite inconvenient for simple function calls, most primitive types implement the *Copy* trait that will automatically pass the variable by-value instead of moving it into the called function. This is both convenient and an optimization for all types whose memory representation does not exceed the pointer size of the platform.

Pointer types. To allow for dynamic resource sharing of complex data types, Rust has a number of *pointer types* that implement traits like *Send* or *Sync* [105] and provide reference counting and mutual exclusion for the data type they encapsulate [106]. This is where the *Drop* trait comes in handy, as it allows for handling of any custom behavior when a variable of that type goes out of scope, like decreasing a reference count and freeing the encapsulated resources when it hits zero.

Implementing primitives. As the guarantees are not provided by the hardware, they need to be implemented in the Rust compiler and runtime. The *unsafe* keyword allows the use of primitives that do not adhere to these safeguards in a transparent and documented way [107].

Foreign Function Interface

A foreign function interface consists of a way to adhere to the foreign calling convention, a way to interface with the foreign binary and a native representation of foreign primitive data types.

Rust has additional attributes to `extern` blocks to provide its FFI [108]. Because of its prevalence, relative portability and susceptibility to errors with severe safety implications, we will focus on the interface to C, which is also the default.

A link attribute like `#[link(name = "owfat")]` on the external API declaration instructs the linker which external library to link to, similarly to a `-lowfat` linker flag in C development.

A comparison of disassembled binaries shows that the resulting code is equivalent to calling a dynamically linked function in C, in that it puts the function arguments on the

stack according to the C calling convention and calls the runtime linker to resolve the called function.

To guarantee compatibility, a `#[no_mangle]` attribute is provided to enforce compatibility for Rust structs and functions and the external *libc* crate provides Rust definitions for common C data types.

2.3.2 Sandboxing in Linux-based Operating Systems

Although the implementation of a suitable sandbox is out of the scope of this work, it is important to know the mechanisms available to build a viable interface to a suitable sandbox.

Namespaces

Namespaces are the prevailing abstraction for sandbox implementations on Linux. Since version 3.8, Linux enables virtualisation of all major subsystems and makes the creation of new namespaces available to userland without requiring superuser privileges [109]. The *namespaces(7)* manual page (available online at [110]) provides an concise overview of the interface and the available namespaces.

The complexity of the Linux API makes it hard to abstract completely, in fact the abstraction has even introduced new vulnerabilities [111]. While it is comprehensive enough to offer a usable replacement for prior incomplete sandboxes built around ACLs, it is not powerful enough to implement advanced mechanisms as the Arbiter [78] memory management, a limitation not shared by more thoroughly abstracted μkernel designs.

System Call Filtering

As mentioned in Section 2.2.2, application-specific sandboxes often use Seccomp-BPF to impose further restrictions on sandboxed processes. It is important to note that the interface described in the *seccomp(2)* manual page (available online at [112]) can also be used with normal user privileges, making it possible to create a comprehensive sandbox from an unprivileged user space application in modern Linux systems.

This chapter shows that the features of recent programming languages and operating systems have not been used in prior research. In the next chapter, we will introduce the design of a compartmentalization solution that uses these features to integrate tightly with the existing development ecosystem.

3 Design

Those who build walls are their own prisoners.
— Ursula K. Le Guin

As we have seen in the previous chapter, existing solutions to sandbox untrusted components in monolithic software fall short of convenient programming interfaces. A major redesign of the software's architecture would require the very programming effort that is the reason why most of today's software does not apply privilege separation in the first place. The solutions that do not require major software changes lack integration into the application development toolchain: They require the installation of external software and external policy specification, or source to source translation of application code. The main design objectives are therefore to automate the *mechanisms* of separation using the existing development ecosystem and to reduce the programming effort to a use of a minimal API. These objectives shape a number of trade-offs in the design.

3.1 Protection Goals

In Section 2.1.1, we have defined an unsafe software component as one which may cause catastrophic consequences to its users, although often, the impact is rather unspectacular. Under the assumption that this un-safety is restricted to a specific part of the program (the unsafe library), it makes sense to treat this part as an outside threat to the security of the rest of the program and to analyse what protection goals an automatic sandboxing solution should enforce against the unsafe component. We will again follow the definition of Avizienis et al., where *security* is a composite of *availability* for service (only) for authorized users, *integrity* as the absence of unauthorized modification and *confidentiality*, „the absence of unauthorized disclosure of information“ [6, p. 13].

Preserving the integrity of the program is the primary motivation for this work. Once set up, it is the role of the sandbox to contain unsafe behavior as a result of a critical error to the unsafe component. The prerequisites and limitations of sandbox isolation are discussed in Section 3.4.3 below.

There is a second form of integrity meaning „absence of improper system alterations“ [6, p. 13], which cannot alone be guaranteed by the sandboxing mechanism: The state changes in the main program through the unsafe function results are intentional and authorized, and their integrity is not covered by a security mechanism against outside threats. Ensuring the validity of the returned data structures is a core task of the compartmentalization mechanism, specifically the Inter-process communication (IPC) discussed in Section 3.4.2.

Integrity in the sense of semantically correct results seems impossible to achieve: Because the unsafe component's computation results are unknown, an improper result

can not generally be detected. Similarly, from the perspective of the user of a library interface, it may seem impossible at first to guarantee *availability* and *reliability*, as the unsafe component is necessary for computation. The application may employ a time out / restart mechanism for availability and reliability. Similarly, integrity may be achieved by consistency checks or redundant computation using different unsafe implementations of an algorithm. The separation provided by Sandcrust can be utilized to accomplish these protection goals in cases where before, an unsafe component may have left the program in an unsafe state, unable to utilize such safeguards within its protection domain.

The last property, *confidentiality*, is difficult: On the surface, the sandbox guarantees confidentiality of the information trusted to the unsafe component by limiting the unsafe component's interaction with the safe program to desired computation results. However, side channel attacks as described in [113] are extremely difficult to defend against, even cryptographic hardware isolation on the CPU itself like Intel's SGX is not beyond them [114]. Moreover, the unsafe component may be able to use its legitimate results to leak information. This limitation is shared by many separation mechanisms with a focus on integrity over confidentiality like Capscium [48], and even prominent data-centric work, which, as remarked by Bittau et al. „[...] allows untrusted code to observe sensitive data, but without sufficient privilege to disclose that data [...]“ at the price of „[...] heightened concern over covert channels, and mechanisms the programmer must employ to attempt to eliminate them“ [77, p. 309].

Sandcrust offers no defense against side-channel information leaks for data explicitly trusted to the unsafe component, but provides a way to control the setup of the sandbox such that it can be placed before making sensitive data available in the main program. Sandcrust does guarantee confidentiality for all data whose access handles are introduced in the main program after sandbox initialization, presuming a sandbox following the specifications detailed in Section 3.4.3.

3.2 Threat Model

The protection goals show that Sandcrust's design falls into the protection model of restricting an untrusted component as defined in Section 2.2.3. We now detail a threat model that informs the design described in the next sections.

1. At the core lies the observation that memory safety guarantees of modern programming languages like Rust do not extend to any legacy libraries they may interface with; to the contrary: in a single protection domain, a security exploit in a legacy component is generally able to subvert the whole program, including its ostensibly safe parts.
2. The next observation is that many security vulnerabilities happen in libraries that in normal operation do not use communication primitives or change data structures outside the arguments and return values of the functions they provide and potentially state changes in the memory of their protection domain.[1]

[1] This is not accurate in the strict sense, because execution has an effect on shared caches and it may be desirable to allow system functions which change internal kernel state.

3. Under that assumption, unsafe libraries will work in a sandbox that revokes all privileges but memory management, communication with the original program and a number of system calls deemed safe.[2].
4. Given such a sandbox, a subverted library can interact with the original program in the following unintended ways:
   (a) crash or exit unexpectedly
   (b) hang on input / invocation
   (c) return no return value / output parameter values via IPC
   (d) return a false number / false types of / corrupt output parameters and/or return value via IPC
   (e) return syntactically correct but semantically wrong results
5. A Trusted Parser on the side of the original program can handle cases (c) and (d) without compromising the type system in the rest of the application. While it will fail to gather appropriate results, it is assumed to do so in a controlled way.
6. Cases a, b and e can be handled in an application-specific way as described in Section 3.1.
7. Everything not sandboxed by the system is considered trusted.

A critical reader may object to the trust by definition expressed in the last step, but the security guarantees given by the high-level language form the basis for the distinction between trusted and untrusted components in this work. Clean-slate designs like in Minix 3 [116] offer much more fine-grained protection domains, but they exclude much of the problem space by demanding a (manual) program redesign to a specific structure.

Note that the implementation of a sandbox as defined in step 3 is outside the scope of this work. Extending the assumptions in step 2 for cases where a library needs to perform IO to serve its purpose would easily be possible in a more holistic implementation, but the interface to a (hypothetical) sandbox in the prototype lacks a way to specify more fine-grained privileges.

3.3 Mechanism Placement

The focus of mechanism placement is on ease of deployment as a solution towards the issue raised by Watson et al.: „Finally, it is clear that the single largest problem with Capsicum and similar approaches is programmability: converting local development into de facto distributed system development hampers application-writers. Aligning security separation with application structure is important as well, if such systems are to mitigate vulnerabilities on a large scale: how can the programmer identify and correctly implement compartmentalizations with real security benefits?“ [48, p. 103]

The study of previous systems with the same protection model in Section 2.2.3 makes it clear that they suffer worst from the intertwined data structures characteristic for monolithic software in the same protection domain. Hence, we will approach componentization with a focus on handling these data structures: Codejail [100] has gone a

[2] While most of the system calls listed in the *syscalls(2)* manual page amount to unauthorized system modifications, there are some legitimate calls like *nanosleep(2)* that a sandboxed program may perform [115].

long way in achieving automatic separation, but with an external system to transform an abundance of annotations of data structures and unsafe functions. Most of these annotations are necessary, because of a problem already described many years earlier by Brumley and Song: „We provide wrappers for common privileged calls instead of automatically generating them from the source because we may not know statically how to wrap a pointer argument to a call. Wrapping pointers requires knowing the pointer's size. Generally functions that take a pointer argument also take an argument indicating the pointer's size. Finding this out is easily done by a human, say by consulting the appropriate man page, but is difficult to do with static analysis alone“ [83, p. 9].

The most obvious starting point to improve on this is the Rust compiler itself. Let us examine if this limitation could be lifted: If the compiler was to track the size of any pointer, it would need to keep track of size of the data pointed to for any Rust method that returns a pointer. This may work for standard library types and methods like this:

```
let greet = "hello";
let hell = &greet[0..4];
let ptr_to_hell = greet.as_ptr();
```

Yet, a slightly more complex case like

```
let boxed_slice = (Vec::<i32>::with_capacity(10)).into_boxed_slice();
let slice_ptr = boxed_slice.as_ptr();
```

is unpleasant already (`slice_ptr` is *0x1*, the length of the slice is 0).

Lastly, external crates may return raw pointers, like the `ptr()` method in *memmap*'s API[3], that such a mechanism will fail to obtain the size for.

Even for standard library types, a safe mechanism would need to convert pointers to a *Sized* representation like a *slice* dynamically, with great care to detect all corner cases correctly, a fragile detection at best, because it relies on unstable internal interfaces. This detection would be part of the TCB, without enjoying the rigorous testing of the standard Rust compiler.

All things considered, the complexity of such a mechanism is not justified, because it does not solve the problem of providing a safe interface around unsafe functions entirely: In addition to the general un-safety of raw pointer dereference that a direct re-use of C function interfaces would propagate into the Rust part of the program, other sources of unsafe behavior such as the blending of error codes and return values in C cannot be handled without defining a wrapper function, a task which can partially be automated using *rust-bindgen*[4]. This further restricts the usefulness of the compiler-based approach to rare corner cases.

If a compiler modification would not yield improvements that justify diverging from the mainline implementation, we should evaluate if a solution can be integrated into the exiting development ecosystem: The problems faced by previous research make it clear that a solution needs to be placed at a level of abstraction where the API of an untrusted component is still accessible, i.e. where the Abstract Syntax Tree (AST) of the program can be manipulated without resorting to external source to source

[3] `https://docs.rs/memmap/0.5.2/memmap/struct.Mmap.html#method.ptr`
[4] `https://github.com/servo/rust-bindgen`

transformation. Another advantage of transformation on the AST level is that the FFI introduced in Section 2.3.1 is readily available; as are data transformation frameworks that will be covered in Section 4.2.

In an existing Rust development setup, a program's AST can be manipulated from two places: from macros and from compiler plugins. Rust macros match on an input of Rust language tokens and place a valid transformation back into the AST [117]. As we will see in Section 4.1, they are subject to a number of limitations, but they have the advantage of being part of the stable Rust interface.

Compiler plugins come in two forms: as *Syntax extensions* and *Lint plugins* [118]. Syntax extensions are a more powerful form of AST manipulation that can run Rust code instead of just matching patterns. Lint plugins extend the correctness checks of the Rust compiler, but are unable to manipulate the AST.

Rust has a module system that provides *crates* to package a library of Rust functionality [119]. Both conventional macros and syntax extensions can be packaged in a crate and used in a program. But there is a caveat: To ensure orderly development of Rust as a relatively young language, Rust has the concept of *stable*, *beta* and *unstable* language and standard library features that are supported by the respective version of the Rust compiler and runtime. To function, syntax extension compiler plugins need access to Rust's internal compiler API, which will not stabilize to enable future development of the compiler [120]. Therefore, compiler plugins cannot be used in stable versions of Rust. To exhaust compatibility with existing Rust implementations, macros were chosen as the interface to the program's AST. A basic example of macros is shown in Listing 3.1. We will discuss the inner workings of Rust macros in more detail in Section 4.1.

```
1  macro_rules! instrument_function {
2      (fn $f:ident($($x:tt)*) $body:block ) => {
3              fn $f($($x)*) {
4                  println!("now we will increment a");
5                  $body
6              }
7      }
8  }
9
10 instrument_function!{
11     fn inc_arg(a: &mut i32) {
12         *a += 1;
13     }
14 }
```

Listing 3.1: A basic macro example

3.4 Protection Domain Separation

The overview in Table 2.1 on page 9 has shown that the overwhelming majority of the related work aiming to restrict untrusted components uses system processes as the

boundary between protection domains. For seamless integration, we are bound to the sandboxing primitives provided by the OS. The prototype is implemented on Linux, for which, as outlined in Section 2.3.2, they are also based on processes as the protection boundary.

While the Linux kernel's large TCB has a relatively large attack surface, it allows for evaluation on a system that is widely used from mobile and embedded consumer devices to large data centers. In addition, the native, non-Rust Linux API interfaces used in the prototype adhere to the POSIX standard[5], which many competing operating systems adhere to to a sufficient degree, so that the prototype should work on the majority of commodity OS's, including Microsoft Windows 10, which has gained a Linux-compatible system call interface [121].

This section builds on a privilege separation model and IPC requirements to form the basis for the requirements for an external sandboxing solution.

3.4.1 Privilege Separation Model

Starting from a macro that transforms the AST of a program, the desired outcome is a program that spans across at least two different processes, one of them in a sandbox. On Linux, new processes can only be spawned with the *fork* and *clone* system calls, although the current standard library fork function internally calls clone [122]. To assume different roles, the code needs to check the return value of the respective library function. At this point, it makes sense that the trusted program resumes execution in the initial process (the *parent*) while the untrusted part should be run in the newly spawned *child* process.

Watson et al. have noted „[...] that libraries cannot create and manage worker processes without interfering with process management in the application itself - unexpected process IDs may be returned by wait()“ [48, p. 99]. This issue is deemed minor because Rust's native `std::process` module does not implement a nonspecific wait function. Instead, `wait()` is a method of a `Child` struct representing a specific Process ID (PID) to call the *waitpid* system call [123] on.

There are three possibilities how the child process can transmute into a sandboxed process, illustrated in Figure 3.1:

1. Execute a separate binary. The forked child process would run one of the library function interfaces to the *execve* system call to replace the current process image with a specified binary, optionally passing arguments or setting its environment. The first advantage of this method is that memory and some other resources initially shared between parent and child process will cease to available to the child process [124]. Second, it is possible to invoke an external helper program such as one of the external sandbox implementations mentioned in Section 2.2.2, to provide a generic setup before they in turn invoke the new binary. The downside of this approach is that it fundamentally breaks

[5] They are necessary, because Rust offers no built-in interface to clone/fork a process:
`https://github.com/rust-lang/rust/issues/6930`
The relevant specifications are:
`http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html`
`http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html`
`http://pubs.opengroup.org/onlinepubs/9699919799/functions/getpid.html`
`http://pubs.opengroup.org/onlinepubs/9699919799/functions/getppid.html`

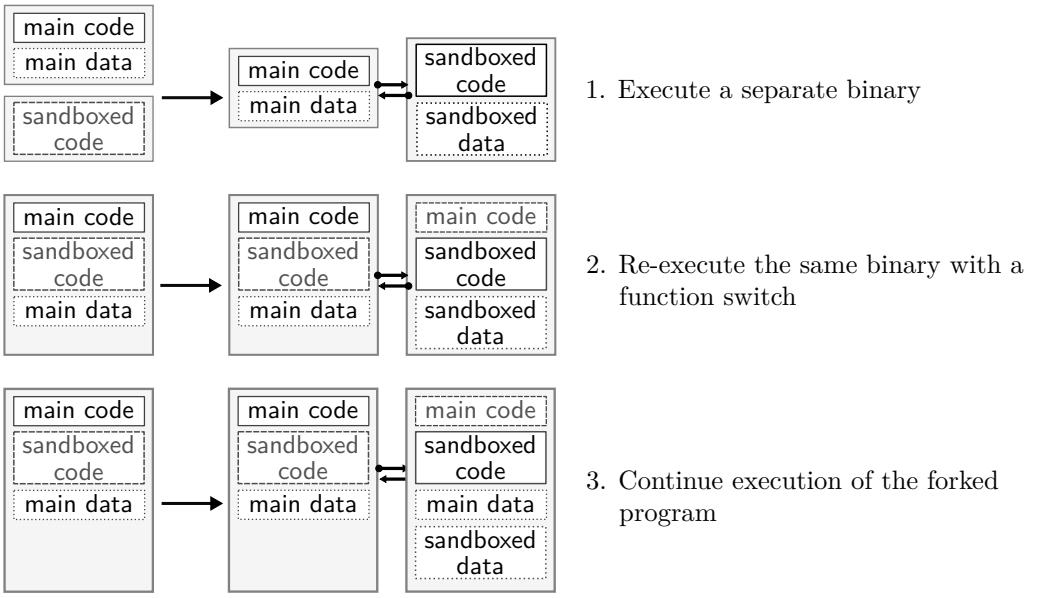


Figure 3.1: Privilege separation alternatives

integration with the program's toolchain, because a second binary needs to be built, packaged and shipped; or that the original process needs to write out the binary prior to invocation, which is cumbersome and may not even be possible, because execution of software residing in user-writeable locations is discouraged for security reasons and might be disabled on the target system.

2. Re-execute the same binary with a function switch. To get around the limitation of the first approach, a function switch could be built into the binary, that, depending on an environment variable or an invocation argument, starts executing the code meant to run in the sandboxed process. Again, an external wrapper program can be used to assist in setting up the sandbox. This is the approach taken by the Chromium browser, where the main binary takes on different roles depending on an invocation argument such as `--type=renderer`. To its disadvantage, this requires altering the control flow of the program's main function, where a different solution may be able to only affect parts of the program that actually invoke an untrusted library function; and this modification is impossible to inject with the macro interface chosen in Section 3.3.

3. Continue execution of the forked program. As the other two options were not feasible for a low-friction integration via macros, the remaining option was to resume execution and set up the sandbox in the forked child process. Besides the implications for the sandbox interface, the key ramification is that with a few exceptions listed in [122], any resources available to the trusted main program at the time of the fork are also available to the child process unless it rescinds the access explicitly. We will discuss the impact of this in Section 3.4.3. In another consequence of this design, Sandcrust is unable to defend against any unsafe behavior stemming from failures in the library initialization. For any non-malicious library that does not process input (e.g. reading a

configuration file) as part of its initialization, unsafe behavior is not possible, because the execution path is static and thus the same as in any working use of the library.

3.4.2 Inter-Process Communication

We will now briefly introduce the IPC design, because it affects the sandbox requirements. The general requirements of an application distributed between at least two processes can be spelled out as follows:

1. Update any state that may have changed in the trusted part of the program and is required for executing a part of the program (usually a function) in the sandboxed process (function arguments and mutable global variables of the library).
2. Request execution of a specific function in the sandboxed process.
3. Replicate any relevant changes in state (i.e. writeable foreign global variables, function results and output parameters) as result of the RPC in the sandboxed process in the trusted process and resume execution.

Sandcrust offers a choice of shared memory (SHM) and pipes for transmitting data, but pipes are always used as a portable RPC synchronization mechanism. We will contrast the IPC performance for both options in Section 5.4. In Unix-like OS's, a pipe is a unidirectional data stream provided by the OS and represented by a read and a write file descriptor. This implies that it needs to be possible to convert any data type to a stream, specifically, the IPC implementation can only work accurately on data types with a known structure and size and will fail to work on pointers to unknown data structures. The consequences of this limitation are evaluated in more detail in Section 5.2, but for now it shall suffice to say that the wrapper placement discussed in Section 3.5.1 makes this less of a concern as it may appear. From the data flow described above, it follows that two pipes are needed to pass state in each direction.

3.4.3 Sandbox Prerequisites

We can now detail the requirements for an external sandbox. From the privilege separation model in Section 3.4.1 follows that the sandbox needs to be set up from within the trusted program, i.e. it needs to be available as a library. To satisfy the assumption laid out in step 3 of the threat model in Section 3.2, a sandbox implementation would need to restrict system calls to [6] :

1. Read and write from/to the respective ends of the IPC pipe file descriptors.
2. In-process memory management (i.e. the `brk` system call).
3. The `exit_group` system call for volunatary termination.
4. Select safe system calls like `nanosleep`[7].

[6] On Unix-like OS's other than Linux, the sandbox additionally needs to allow the *getppid()* system call. This is necessary to provide automatic termination of the sandboxed process when the parent process terminates. On Linux, this is set up before sandboxing via *prctl()*, but because POSIX lacks a generic solution to the problem, the generic implementation spawns a thread to poll for a change in the parent process ID.

[7] Some safe system calls (especially *gettimeofday(2)*) are actually handled via the virtual dynamic shared object (vDSO) [125] and therefore always available.

Depending on the nature of the untrusted software, more privileges like reading random data from `/dev/urandom` may be required; or it may be desirable to allow writes to `stderr` to print error message originating from the untrusted code. Sticking to the assumption laid out in step 2 of the threat model, we limit the discussion to libraries without the need for special privileges. In that scenario, the required interface is a library function that, given the pipe file descriptors, sandboxes the calling process by applying the restrictions outlined above. Beyond prototype design, this interface would need to be extended to accommodate additional sandboxing policies or allow for a choice of sandboxing solutions. With the current lack of any turn-key sandboxing solution in Rust, the design of such an extension is a purely theoretical exercise and therefore left out of the prototype implementation.

We have introduced *Seccomp-BPF* as a suitable solution to enforce these restrictions in Section 2.3.2 and trust that a suitable interface can be constructed as a library, potentially relying on an existing abstraction such as *libseccomp*[8].

In-memory resources are outside of the scope of a system call based sandboxing mechanism. From Shankar and Wagner's overview [126, p. 2269], the following in-memory resources deserve special consideration:

- Process memory mappings
- Shared memory segments
- The Unix environment

Both process memory mappings and shared memory segments may have implications beyond the danger of information leaks discussed in Section 3.1, as the sandboxed process may use them to compromise the integrity of the trusted program. Sandcrust has no access to information about the memory layout at compile time, but offers explicit management of the sandboxed process as a way to control the resources available in program memory a the time of the fork.

This control can be extended to the Unix environment in a limited fashion: It is effective if the sandboxed process is initialized before the introduction of (potentially sensitive) environment variables. While the `**environ` data structure can be made inaccessible using *clearenv(3)* [127], the environment inherited from the caller of the main program still resides in the memory of the sandboxed process. The inherited Unix environment could therefore also be subject to information leaks, which is a limitation of the sandbox isolation.

3.5 Sandcrust Workflow

With all major design decisions in place, this section will explain the workflow of the prototype. First, the requirements for compartmentalization are summarized, before explaining the application flow and the API.

The stripped down example program shown in Listing 3.2 will serve as an illustration.

The example uses the *memset(3)* and *time(2)* functions from the C standard library. The `extern` block in line 4 introduces the function declarations to Rust, automatically

[8] `https://github.com/seccomp/libseccomp`

```
1  extern crate libc;
2  use libc::{time_t, c_int, size_t, c_void};
3
4  extern {
5      fn memset(buf: *mut c_void, c: c_int, n: size_t) -> *mut c_void;
6      fn time(time: *mut time_t) -> time_t;
7  }
8
9  fn wrap_memset(buf: &mut Vec<u8>, fillnum: u8) {
10     let pbuf = buf.as_mut_ptr() as *mut c_void;
11     let buflen = buf.len() as size_t;
12     let c = fillnum as c_int;
13     unsafe {
14         memset(pbuf, c, buflen);
15     }
16 }
17
18 fn wrap_time() -> u64 {
19     unsafe {
20         let timep: *mut time_t = std::ptr::null_mut();
21         time(timep) as u64
22     }
23 }
24
25 fn main() {
26     let mut buf = vec![0u8; 256];
27     let fillnum: u8 = 161;
28     wrap_memset(&mut buf, fillnum);
29
30     let seconds = wrap_time();
31     println!("Seconds passed: {}", seconds);
32 }
```

Listing 3.2: Original API example program

marking them as *unsafe*, because it defaults to C language bindings. Wrapper functions `wrap_memset` and `wrap_time` are responsible for encapsulating the unsafe behavior and translating between native Rust data types and C types imported from the *libc* crate. The most notable conversion is that of the `Vec<u8>` byte vector to a void pointer in line 10. Note that in Rust, it is safe to create, but unsafe to dereference a *raw pointer*.

3.5.1 Compartmentalization Requirements

Necessary Tasks

The requirements for automatically sandboxing an unsafe component can be summarized as follows:

- Set up a sandboxed protection domain.
- Provide an IPC mechanism.
- Implement an RPC endpoint.

- Update relevant program state (e.g. global variables of the library), if applicable.
- Marshall / unmarshall arguments and perform RPC calls.

Depending on the library, the constructed protection domain needs to be stateful in order to preserve internal state of the unsafe library between library function calls.

Wrapper Placement

A close look at the example shown in Listing 3.2 reveals that calling functions through the FFI requires `unsafe` blocks, a feature described in Section 2.3.1, suspending the guarantees of the Rust language to deal with the unsafe pointers pervasive in C. It is therefore advisable to sandbox the wrapper function instead of the original C function call. A second reason is that the wrapper functions transform idiomatic Rust data types to the Rust representations of C data types in the FFI function declaration, and some of these transformations may have safety implications, such as the `NULL` pointer passed to `time()` in line 21 of the example program.

3.5.2 Componentized Application Flow

The current implementation of Sandcrust provides two modes of operation which are detailed hereafter, before discussing the API available to the programmer in the next section. For variability of use, both a single-invocation and a stateful version of the sandboxing automation have been implemented. We will first discuss the application flow, illustrated in Figure 3.2 for a single-invocation componentization and then contrast it with the stateful sandboxing implementation.

Sandcrust for Single Function Invocations

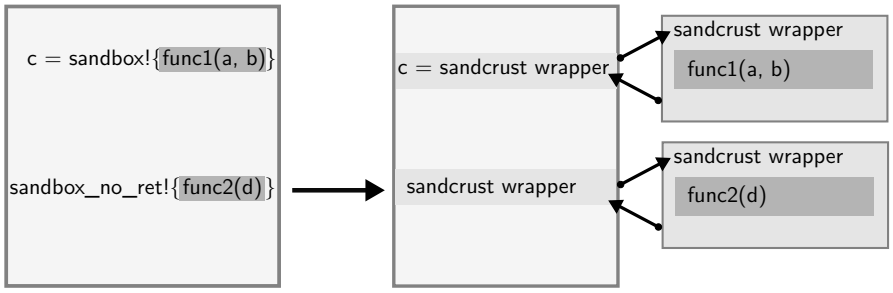
Annotating a function invocation replaces the original function call with code which

(a) Forks off a second process, for reasons described in Section 3.4.1, which:
  (i) Sets up a sandbox, following the requirements set out in Section 3.4.3.
  (ii) Runs the wrapped function.
  (iii) Transfers output parameters and any return value back using IPC.
  (iv) Exits.

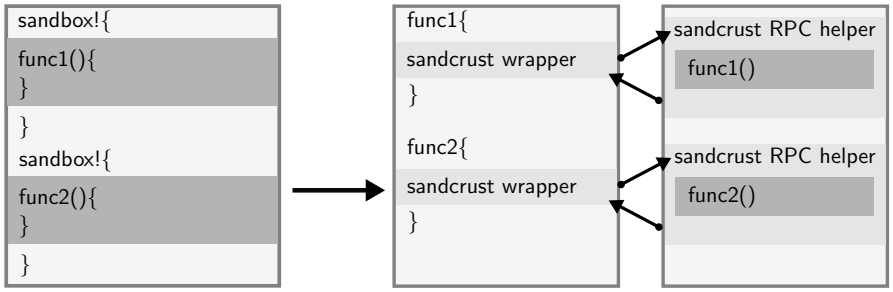
(b) Incorporates the changed values back into the main program.
(c) Collects the child's exit status and resumes the main program.

The advantage of this approach is its simplicity, but besides the need to annotate every function call, there are other drawbacks:

1. A new sandbox is spawned for each function call, causing overhead.
2. Internal state of a called library is not preserved between calls.
3. Any missed invocation will execute in the protection domain of the trusted program.
4. Because of sandbox originates from a forked process, it inherits all data present in the main program at the time of invocation.



(a) Sandcrust for single function invocations



(b) Sandcrust for stateful sandboxing

Figure 3.2: Sandcrust overview

Sandcrust for Stateful Sandboxing

To overcome the limitations of the "fire and forget" approach, the stateful sandboxing implementation works by annotation of the function *definition*. This transforms the function in a way that it executes normally in the sandboxed process, but invokes a wrapper function in the main program that will:

(a) Initialize a persistent sandboxed process in line with Section 3.4.1 if necessary by setting up IPC as described in Section 3.4.2, forking off a process which is then sandboxed and runs an RPC endpoint.
(b) Perform an RPC.
(c) Update any relevant state in the sandbox, as it may have changed since sandbox initialization.
(d) Collect the results and continue execution.

In the sandbox, a loop waits for RPCs from the main program, calling a helper function that:

(i) Applies the state updates in lockstep with the main program.
(ii) Runs the wrapped function.
(iii) Transfers writeable foreign global variables, output parameters and any return value back using IPC.

We will revisit the details of the implementation in Section 4.3.

3.5.3 Sandcrust API

After discussing the internal mechanisms, we now present the API available to the user of Sandcrust.

The first step in using Sandcrust in a project is to declare dependency on it in the project's *Cargo.toml*. Next is importing the Sandcrust crate into the program, including macros:

```
4  #[macro_use]
5  extern crate sandcrust;
6  use sandcrust::*;
```

With these prerequisites in place, let us now explore the different ways to use Sandcrust.

Sandcrust for Single Function Invocations

Once Sandcrust is set up, we just need to annotate the wrapper functions' invocations in our example from Listing 3.2 to run each in an ephemeral sandbox:

```
32  sandbox_no_ret!{wrap_memset(&mut buf, fillnum)};
```

```
34  let seconds: u64 = sandbox!{wrap_time()};
```

Because this mode of usage is intended for sandboxing a single function *invocation*, this is where the annotation in form of a Rust macro has been placed. Annotations for functions with and without return value differ, because the type of return values can not be inferred automatically. The reason for this limitation is that the function declaration is not known to the sandboxing mechanism.

Sandcrust for Stateful Sandboxing

The stateful sandboxing implementation works quite differently, compared to the single invocation API. The modified example in Listing 3.3 shows that the annotations have moved to the function definition, eliminating the need to annotate each invocation and enabling return value detection.

```
extern crate libc;
use libc::{time_t, c_int, size_t, c_void};

#[macro_use]
extern crate sandcrust;
use sandcrust::*;

extern {
    fn memset(buf: *mut c_void, c: c_int, n: size_t) -> *mut c_void;
    fn time(time: *mut time_t) -> time_t;
}

sandbox!{
fn wrap_memset(buf: &mut Vec<u8>, fillnum: u8) {
    let pbuf = buf.as_mut_ptr() as *mut c_void;
    let buflen = buf.len() as size_t;
    let c = fillnum as c_int;
    unsafe {
        memset(pbuf, c, buflen);
    }
}
}

sandbox!{
fn wrap_time() -> u64 {
    unsafe {
        let timep: *mut time_t = std::ptr::null_mut();
        time(timep) as u64
    }
}
}

fn main() {
    let mut buf = vec![0u8; 256];
    let fillnum: u8 = 161;
    wrap_memset(&mut buf, fillnum);

    let seconds = wrap_time();
    println!("Seconds passed: {}", seconds);
}
```

Listing 3.3: Example program modified for stateful sandboxing

Advanced Usage of the Stateful Sandbox

Sandcrust for custom data types. It may be necessary to annotate custom data types, for *custom derive* (described in more detail in Section 4.2):

```
#[derive(Serialize, Deserialize, PartialEq)]
struct CustomStruct {
    x: f32,
}
```

Explicit Sandbox management and respawn. To counter the issue of leaking sensitive information raised towards the end of Section 3.1, it is possible to initialize the persistent sandbox explicitly:

```
fn main() {
    sandcrust_init();
    let secret = "[redacted]";
    [...]
}
```

To explicitly terminate the sandbox there is `sandcrust_terminate()` as an antithetic function. Any attempt to run a sandboxed function after Sandboxed termination will result in controlled program termination, unless the `auto_respawn` compile-time feature is used, in which case a new persistent sandbox is spawned automatically.

Together, `sandcrust_init()` and `sandcrust_terminate()` can be used to shield different sets of sandboxed functions from each other, although the prototype implementation only supports one sandbox at a time.

Setting the maximum data transfer size for SHM. If IPC over SHM is enabled via the `shm` compile-time feature flag, it may be necessary to adjust the size of the shared memory region from the default exported via `SANDCRUST_DEFAULT_SHM_SIZE`. To that end, the Sandcrust API provides a specialized initialization function and an independent setter function:

```
sandcrust_set_shm_size(512);

sandcrust_init_with_shm_size(512);
```

The changed size only comes into effect after a new sandbox in initialized.

Mutable foreign globals. Sandcrust has support for mutable foreign global variables in the form of the `sandcrust_wrap_global` annotation:

```
sandcrust_wrap_global!{
    #[link(name = "linkname")]
    extern {
        static mut variable1: ::libc::c_int;
        static mut variable2: wrapable_type;
    }
}
```

This annotation is currently restricted to one extern block with a link attribute and a list of `static mut` declarations. This mainly limits the support to one external library with mutable variables, but may enable important use cases, as experienced by Watson et al.: „In adapting gzip, we were initially surprised to see a performance improvement; investigation of this unlikely result revealed that we had failed to propagate the compression level (a global variable) into the sandbox, leading to the incorrect algorithm selection“ [48, p. 101]. Immutable global variables are not in need of special handling and the use of mutable global variables in native Rust is *unsafe* and should be avoided in the implementation of Rust wrapper functions for unsafe C library functions.

After this introduction of the abstract interface, the next chapter discusses the prototype implementation.

4 Implementation

Why do I not wake up with the gin already in me
— Matthew Garrett

This chapter sheds light on challenges encountered during prototype implementation and thereby illustrates the inner workings of the prototype outlined in Chapter 3. After a brief introduction of the Rust macro system in Section 4.1, we will first revisit the handling of function argument and return value types in IPC in Section 4.2 and finally depict the challenges in creating a stateful sandbox out of simple annotations of a monolithic program in Section 4.3.

4.1 Metaprogramming with Macros

Section 3.3 has explained the rationale for basing the prototype implementation on Rust's macro system. The idea to use macros was born out of the realization that a compiler plugin would never work in stable Rust, and the assumption that metaprogramming in the macro system might leverage sufficient changes to the AST to implant sandboxing in the program just from the macro's input. This implicates that monomorphization, the generation of specialized code from a generic template, would have to be enough. Thanks to Rust's many ways of code reuse, of which Alexis Beingessner has written an excellent overview on his blog [128], this is actually true albeit challenging, as we will see in the following sections. Before, we will start with a basic introduction to the macro system (a full introduction is found in [117] and a detailed description in Daniel Keep's extremely helpful *The Little Book of Rust Macros* [129]).

We have mentioned that Rust macros match and transform language tokens in the AST. Given an identifier, the `double` macro in Listing 4.1 naïvely assumes the identified object implements the `Add` trait and doubles it. When given a number of statements separated by "`;`", it prints the value of a freshly defined variable `a` before executing each statement twice.

The order of the two match arms in line 2 and 3 is important, because the variable `a` also matches a statement (`$s:stmt`) and the `let` statement in the match expansion would then be used where an expression is expected in the assignment in line 15. This will fail, because each macro expansion must result in a valid syntax tree. If we execute the program as listed, it outputs the following:

```
1  macro_rules! double {
2      ($i:ident) => { $i + $i };
3      ($($s:stmt);+) => {
4          let a = 42;
5          println!("we've got a as {}, right?", a);
6          $(
7              $s;
8              $s;
9          )+
10     };
11 }
12
13 fn main() {
14     let a = 1;
15     let two = double!(a);
16     println!("double {} is {}", a, two);
17     double!(println!("once more, a is {}", a); println!("really!"));
18 }
```

Listing 4.1: A macro example

```
double 1 is 2
we've got a as 42, right?
once more, a is 1
once more, a is 1
really!
really!
```

The conflicting statements about the value of variable `a` are a result of Rust's *hygienic* macro system, where „macro expansion happens in a distinct 'syntax context', and each variable is tagged with the syntax context where it was introduced“ [117]. Therefore, the `println!()` in line 5 sees variable `a` as defined in line 4, whereas the expanded statements see variable `a` as defined in line 14.

Lastly, the example illustrates that macros are expanded recursively: the macro system replaces a match and continues at the node of the macro, consecutively expanding new macros (like `println!()`). We will see this pattern in action in Section 4.3.3.

4.2 Function Signatures and IPC

As briefly mentioned in Section 3.4.2, the initial design relied on shared memory, which was implemented using a file in Linux' `/dev/shm` RAM file system and mapped using the *memmap* crate[1].

This approach failed, because it was not possible to determine the size for complex data types: Using `::std::mem::size_of_val()`[2] to determine the size and subsequently

[1] `https://crates.io/crates/memmap`
[2] `https://doc.rust-lang.org/std/mem/fn.size_of_val.html`

transfer output parameters via the unsafe `std::mem::transmute_copy()` function[3] worked well for primitive data types, but came to its limits with heap-based data structures, which did not implement the *Copy* trait explained in Section 2.3.1. There is a *Clone* trait, but for example for Vectors, it only clones the vector data structure, but not the heap memory managed by it.

Without a generic mechanism available, the initial macro implementation around function invocations introduced in Section 3.5 would need a compiler intrinsic to get the type of an argument identifier. Such a function exists in the form of `std::intrinsics::type_name()`[4], but `std::intrinsics` is part of the unstable interface, because it relies on internal workings of the compiler. Moreover, each type of the standard library would have required a custom transfer implementation (each adding an unsafe function to the TCB for compartmentalization); and passing custom data types would simply be impossible.

A solution already exists when the problem is approached from a slightly different angle: In the context of IPC, the problem can be paraphrased as a problem of *serializing* and *deserializing* a data structure. Unfortunately, Rust does not offer a built-in equivalent for serialization as exists in form of the special `Send` trait[5], which is automatically assigned by the Rust compiler to types that can be passed between threads. However, an external solution is available in the form of *Serde*[6], a popular *ser*ialization and *de*serialization framework. Compartmentalized Rust software like the Servo web browser[7] uses *Bincode*[8] for encoding of data processed by Serde for IPC, and it has been successfully used to solve the problem.

However, the initial evaluation of IO performance found that Bincode adds a disproportionate overhead for simple data structures. To alleviate the impact on Sandcrust's performance, an optional optimization for byte vectors and slices can be enabled via the `custom_vec` compile-time feature flag. We will contrast the performance to native Bincode encoding in Section 5.4.2.

A simple reimplementation of the IPC mechanism using Bincode over a pipe resulted in code reduction by 45% in the prototype. This is offset by the addition of the Bincode deserializer and its dependencies to the TCB: The trusted part of the program relies on correct parsing of the returned stream, as noted in step 6 of the threat model in Section 3.2. Employing a widely used component in place of a custom solution is deemed beneficial to security. As Meyer and Arnout have noted: „In software design, laziness is a virtue: it's better to reuse than to redo“ [130, p. 23]. The optional use of SHM as a means of communication between processes may make it necessary to adapt the size of the shared memory region to the maximum data size as explained in Section 3.5.3, whereas the stream-based nature of pipes relieves the user from this task.

Serde makes use of Rust's powerful trait system (described in Section 2.3.1) to implement *Serialize* and *Deserialize* traits for all feasible standard library data structures.

[3] `https://doc.rust-lang.org/std/mem/fn.transmute_copy.html`
[4] `https://doc.rust-lang.org/std/intrinsics/fn.type_name.html`
[5] `https://doc.rust-lang.org/reference.html#the-send-trait`
[6] `https://serde.rs/`
[7] `https://servo.org/`
[8] `https://crates.io/crates/bincode`

This restricts function arguments to data types supported in Serde, but for compound data structures, Rust has another trick up its sleeve:

Procedural Macros [131] are a kind of compiler plugin stabilized with the release of Rust 1.15 on the 2nd of February 2017, that allow for *custom derive* of traits by deriving a generic implementation from the AST. This may sound complex, so here is an illustrating example:

```
#[derive(Serialize, Deserialize, PartialEq)]
struct Entity {
    x: f32,
    y: f32,
}

#[derive(Serialize, Deserialize, PartialEq)]
struct World {
    entities: Vec<Entity>,
}
```

Serde's *serde_derive* crate now constructs implementations of the derived traits that are braced in the supplied implementation for basic data types. While annotation of custom data structures is necessary, the advanced features of Rust break it down to the derive attribute, as opposed to the extensive annotations or wrapper implementations required by prior work.

The API laid out in Section 3.5.3 includes two distinct macros for single function invocations, `sandbox!()` and `sandbox_no_ret!()`, and mentioned that this complexity in the API stems from the lack of access to the function declaration. Idiomatic Rust wraps function results in an `Option` enum to indicate the possibility of absence in the form of `Some(T)` for a result type T, or `None`. However, Rust's type checking would (correctly) refuse to assign a possible value of `None` to a result variable. Moreover, for functions with return value, the return type needs to be specified for new variable declarations. This is necessary as soon as different types are passed through Bincode, because then the `bincode::deserialize_from()` function is *virtualized* at compile time, i.e. the compiler expands the generic function definition for any type implementing the `Deserialize` trait to a specific implementation for each data type used. The macro for stateful sandboxing has access to function declarations and is therefore able to provide a simplified interface.

4.3 Implementing a Stateful Sandbox

This section details the implementation of Sandcrust's stateful sandboxing, whose general application flow was introduced in Section 3.5.2. Compared to the simplistic application flow of single invocation wrapping, the stateful sandboxing posed three major challenges to the implementor:

- Managing a global, persistent Sandbox from insular macro transformations throughout the original source code.
- The implementation of an RPC endpoint.
- Ad-hoc generation of appropriate argument marshalling and unmarshalling routines from within the macro interface.

Figure 4.1 shows the call graph for sandboxing a function.

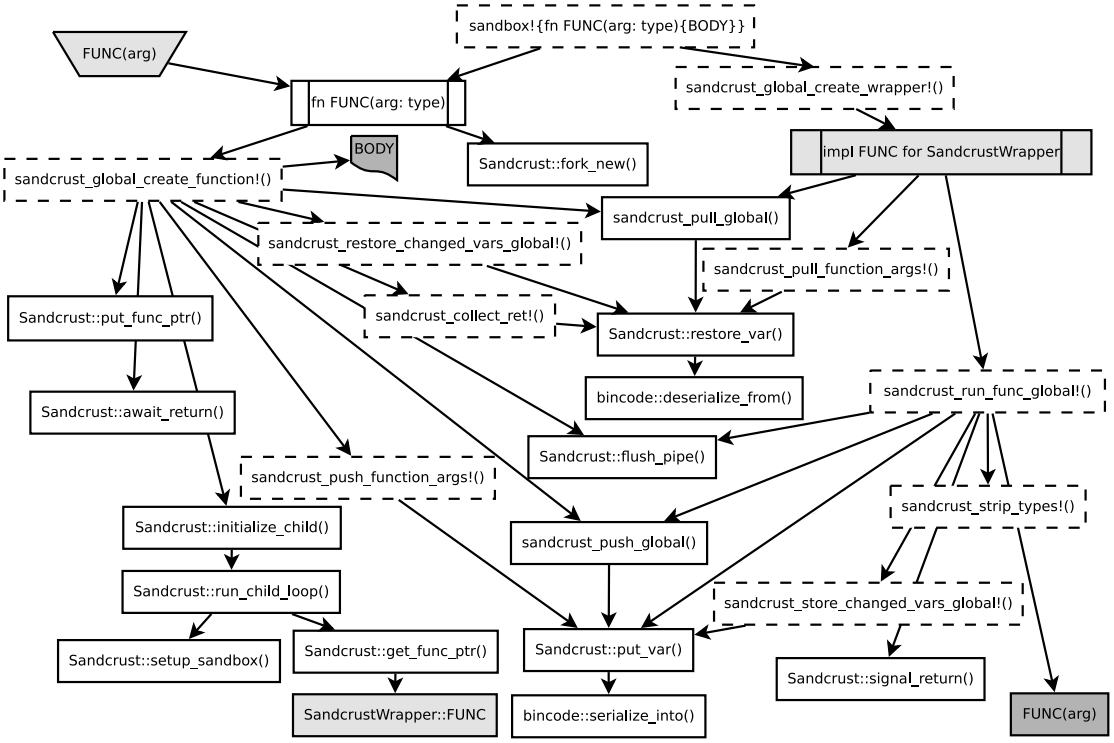


Figure 4.1: Sandboxing a function *fn FUNC(arg: type){BODY}* with Sandcrust using default compile flags results in this call graph. Note that macro invocations are resolved at compile time. Calling *FUNC()* from the main program as illustrated in the top left corner results in an invocation of *FUNC()* in the sandboxed process, depicted in the bottom right corner. Sandcrust modifies the function declaration so that a call from the main program results in an RPC to the *SandcrustWrapper::FUNC()* helper function, which in turn invokes *FUNC()* in the Sandboxed body, where it runs *BODY*.

4.3.1 Managing a Global Sandbox

In a programming language like C++, creating a global object for managing the persistent sandbox would be a straightforward matter of initializing a global variable from a constructor. Rust supports global variables in the form of the `static` keyword, but only allows initialization by a constant expression [132]. Initially, the global structure drew on primitive types as a representation of global structures:

Each invocation of a wrapped function would check the value of the `cmd_send` in line 2 and set up the stateful sandbox, if necessary. Subsequently, it would set up a new `Sandcrust` object with clones of the raw file descriptors derived from the global file

```
1  pub struct SandcrustGlobal {
2      pub cmd_send: std::os::unix::io::RawFd,
3      pub result_receive: std::os::unix::io::RawFd,
4      pub child: SandcrustPid,
5  }
6
7  pub static mut SANDCRUST_GLOBAL: SandcrustGlobal =
   ↪   SandcrustGlobal{cmd_send: 0, result_receive: 0, child: 0};
```

Listing 4.2: Original global data structure

descriptors via the `dup()` function[9], where they would be consumed into native Rust `File` objects, used for IPC, and closed as the File objects went out of scope.

The *lazy_static* crate[10] achieves a direct global representation of a dynamically set up *Sandcrust* object by making ingenious use of Rust's `Deref` trait to set up the global object on first dereference: As can be seen from line 9 of Listing 4.3, the global object is

```
1  pub struct Sandcrust {
2     file_in: ::std::fs::File,
3     file_out: ::std::fs::File,
4     child: SandcrustPid,
5  }
6
7  lazy_static! {
8     pub static ref SANDCRUST:
   ↪   ::std::sync::Arc<::std::sync::Mutex<Sandcrust>> = {
9        std::sync::Arc::new(std::sync::Mutex::new(Sandcrust::fork_new()))
10    };
11 }
12
13 pub static mut SANDCRUST_INITIALIZED_CHILD: bool = false;
```

Listing 4.3: Global Sandcrust object using *lazy_static*

secured by a `Mutex`. The second global object `SANDCRUST_INITIALIZED_CHILD` in line 13 is necessary, because the sandboxed process is forked from a Sandcrust method and thus the Mutex is locked at the time of the fork. It is used in an "`if`" statement in the instrumented wrapper function to distinguish between sandbox- and main program side invocation of wrapped functions. If true, a function invocation executes the original function body. If false, Sandcrust is initialized (if the first call to a sandboxed function) and Sandcrust's RPC mechanism is invoked.

The Mutex provides thread safety, but at the cost of serializing parallel execution in the main program if it uses sandboxed functions concurrently. This is a limitation of the prototype, as its IPC mechanism in its current implementation is unable to handle concurrent RPC.

[9] `http://man7.org/linux/man-pages/man2/dup.2.html`
[10] `https://crates.io/crates/lazy_static`

4.3.2 RPC Endpoint Implementation

Conceptionally, the RPC endpoint is a simple loop. A key problem to solve is the identification of the called function in an incoming RPC request, because the IPC mechanism described in Section 4.2 is not able to serialize a function pointer, and custom derive only works on structs and enums.

The solution is pretty much what any shrewd C programmer would have done in the first place: Cast the function pointer to an array of bytes on the calling side and back to a function pointer in the RPC loop. The calling side is displayed in Listing 4.4.

```
pub fn put_func_ptr(&mut self, func: fn(&mut Sandcrust))  {
    unsafe {
        let func_ptr: *const u8 = ::std::mem::transmute(func);
        #[cfg(target_pointer_width = "32")]
        let buf: [u8; 4] = std::mem::transmute(func_ptr);
        #[cfg(target_pointer_width = "64")]
        let buf: [u8; 8] = std::mem::transmute(func_ptr);
        let _ = self.file_in.write_all(&buf).expect("sandcrust: failed to
 ↪  send func ptr");
    }
}
```

Listing 4.4: Method to transmit a function pointer

And the RPC loop simply reverses the transformation via the `get_func_ptr()` method displayed in Listing 4.5.

```
pub fn get_func_ptr(&mut self) -> fn(&mut Sandcrust)  {
    #[cfg(target_pointer_width = "32")]
    let mut buf = [0u8; 4];
    #[cfg(target_pointer_width = "64")]
    let mut buf = [0u8; 8];
    self.file_out.read_exact(&mut buf).expect("sandcrust: failed to read
 ↪  func ptr");
    let func_ptr: *const u8 = unsafe { std::mem::transmute(buf) };
    let func: fn(&mut Sandcrust) = unsafe { std::mem::transmute(func_ptr)
 ↪  };
    func
}
```

Listing 4.5: Receive a function pointer and transform it back to a function type

This is necessary, because the alternative of using a form of lookup table to match an identifier to a function is not realizable from a macro expansion: Each macro can only provide a local transformation of the wrapped function's AST and has no access to information about the other wrapped functions. On the other hand, a global data structure, even if initialized lazily as described in the last section, has no way to collect information about the compile-time transformations. Therefore the issue had to be solved with information local to the macro transformation.

This example shows that Rust's unsafe features are sometimes necessary for systems programming, but `unsafe` blocks provide a clear indication of code that flouts Rust's memory safety guarantees and contain the parts of the code where safety guarantees are provided by the programmer, instead of the compiler. The last piece of the puzzle is how the generic function signature `fn(&mut Sandcrust)` translates to a call of a specific function and will be examined in the next section.

4.3.3 Argument Handling Routine Generation

It must be borne in mind that all specializations of the Sandcrust library happen through iterations of AST transformations when compiling the target program. Therefore, these transformations must suffice to build a mechanism that enables the stateful sandbox to correctly handle RPCs for all sandboxed functions when, for each function, transformation happens locally without access to the AST of the other sandboxed functions. The last section has introduced a simple mechanism to remotely call any function in the sandboxed process, but provides no notion of what argument handling the function entails. In consequence of this, any sandbox-side IPC needs to be contained in the remotely called function body. This contradicts the notion that the function signature of the wrapped function should remain unchanged, to dispense with any annotations of the function invocations and to allow for recursive calls of a wrapped function within the sandbox.

Obviously, what is needed is a helper function per wrapped function with a generic function signature. One would assume that an AST modifying macro system provided a way to generate new identifiers (e.g. `func_`*wrapped*`()`) and indeed, this is the purpose of the `concat_idents!()` macro. Unfortunately, this macro is only included in Rust's unstable interface and what's more, it is broken for use with new function identifiers[11].

The solution depicted in Listing 4.6 re-uses the identifier by implementing a new trait by the name of the wrapped function for the empty struct `SandcrustWrapper` defined in the library crate. This works because the same-crate restriction on method implementations are relaxed for traits[12].

The newly defined trait function now forms the basis for the `func` pointer transformed in the last section:

```
let func: fn(&mut $crate::Sandcrust) = $crate::SandcrustWrapper::$f;
```

The first macro in the function body in line 8 of Listing 4.6 pulls any global variables, and the second macro in line 9 pulls the function arguments according to the argument list (`$($x)*`) passed into the macro.

We will now discuss an implementation detail of the third invoked macro in line 10, `sandcrust_run_func_global!()`, which runs the function.

[11] This is confirmed by many reports on the Internet of people facing the same problem:
`https://github.com/rust-lang/rust/issues/12249`
`https://github.com/rust-lang/rust/issues/13294`
`https://www.reddit.com/r/rust/comments/3e09gn/macro_concatenate_identifiers/`

[12] For an illustration of the impressive capabilities of Rust traits, see this excellent article by Jonathan Turner:
`http://www.jonathanturner.org/2016/02/down-the-rabbit-hole-with-traits.html`

```
($has_retval:ident, $has_vec:ident, fn $f:ident($($x:tt)*)) => {
    trait $f {
        fn $f(sandcrust: &mut $crate::Sandcrust);
    }

    impl $f for $crate::SandcrustWrapper {
        fn $f(sandcrust: &mut $crate::Sandcrust) {
            sandcrust_pull_global(sandcrust);
            sandcrust_pull_function_args!(sandcrust, $($x)*);
            sandcrust_run_func_global!($has_retval, $has_vec, sandcrust,
↪   $f($($x)*));
        }
    }
}
```

Listing 4.6: Generating a trait function for argument handling

To recap, the `$f:ident($($x:tt)*)` match includes the argument types, for example `func(a: i32, b: Vec<u8>)`. To run the function, the types need to be stripped away to get a function invocation like `func(a, b)`. This is achieved by the `sandcrust_strip_types!()` macro. A straightforward implementation to simply transform the function arguments would look like this: Lamentably, parsing `arg` into an

```
1  macro_rules! sandcrust_strip_types {
2      ($arg:ident : $var_type:ty) => ($arg);
3      ($arg:ident : &mut $var_type:ty) => (&mut $arg);
4      ($arg:ident : &$var_type:ty) => (&$arg);
5      (mut $arg:ident : $var_type:ty) => ($arg);
6      ($f:ident($($arg:expr),+)) => ($f(
7          $(
8           strip_types!($arg)
9          ),+)
10     );
11     ($f:ident()) => ($f());
12 }
```

Listing 4.7: A straightforward implementation of a type strip macro

`expr`ession in line 6 causes the match to become *un-destructible* [129], i.e. any subsequent invocation sees `arg` as one AST node, so the recursive matches in lines 2-5 fail: The "`a: i32`" is one node and does not match `$arg:ident : $arg_type:ty`. This can be avoided by preserving the arguments as a token tree (`tt`), but that in turn breaks the repetition match in line 8.

Instead, Listing 4.8 shows the actual implementation.

A function without any arguments is matched by line 15. Line 14 starts the recursion using *Push Down Accumulation* [133], and starts building up the result after the "`->`". This solves the problem of matching the inner argument list that was unsuccessfully addressed with the broken recursion in lines 6-10 of Listing 4.7. Line 3 matches a function

```
1  macro_rules! sandcrust_strip_types {
2      (($head:ident : &mut $var_type:ty, $($tail:tt)+) ->
   ↪   ($f:ident($($body:tt)*))) => (sandcrust_strip_types!(($($tail)+) ->
   ↪   ($f($($body)* &mut $head,))));
3      (($head:ident : &mut $var_type:ty) -> ($f:ident($($body:tt)*))) =>
   ↪   ($f($($body)* &mut $head));
4
5      (($head:ident : &$var_type:ty, $($tail:tt)+) ->
   ↪   ($f:ident($($body:tt)*))) => (sandcrust_strip_types!(($($tail)+) ->
   ↪   ($f($($body)* &$head,))));
6      (($head:ident : &$var_type:ty) -> ($f:ident($($body:tt)*))) =>
   ↪   ($f($($body)*  &$head));
7
8      ((mut $head:ident : $var_type:ty, $($tail:tt)+) ->
   ↪   ($f:ident($($body:tt)*))) => (sandcrust_strip_types!(($($tail)+) ->
   ↪   ($f($($body)* mut $head,))));
9      ((mut $head:ident : $var_type:ty) -> ($f:ident($($body:tt)*))) =>
   ↪   ($f($($body)* $head));
10
11     (($head:ident : $var_type:ty, $($tail:tt)+) ->
   ↪   ($f:ident($($body:tt)*))) => (sandcrust_strip_types!(($($tail)+) ->
   ↪   ($f($($body)* $head,))));
12     (($head:ident : $var_type:ty) -> ($f:ident($($body:tt)*))) =>
   ↪   ($f($($body)* $head));
13
14     ($f:ident($($tail:tt)+)) => (sandcrust_strip_types!(($($tail)+) ->
   ↪   ($f())));
15     ($f:ident()) => ($f());
16 }
```

Listing 4.8: Type strip macro implementation

with one "`&mut`" argument, strips away the type and outputs a function invocation in place of the macro, with the added argument. Finally, the first match in line 2 performs the strip and recursive invocation, if there are still more arguments in the `$($tail:tt)+` list. Lines 5 till 12 repeat the recursive process for other types of arguments.

After this field trip to the darkest corners of the macro system, we will return to the end user's perspective by evaluating the prototype in the next chapter.

5 Evaluation

There ain't no fate but what we take from the moment, wasted time is the only opponent.
— Sole and the Skyrider Band (Hello Cruel World)

This chapter evaluates the prototype. In Section 5.1, we contrast the integration into the source code of an existing program with Rust language elements that denote nonstandard forms of function execution. Section 5.2 compares the possible interactions between a sandboxed library and the main program to execution in the same address space. Drawing on case studies in Section 5.3, the performance of the prototype is evaluated in Section 5.4.

5.1 Language Integration

Table 5.1 demonstrates possible integrations of a sandboxing mechanism with various Rust language elements. In the following, we discuss the adequacy of each element and assess the suitability of the prototype's syntax by comparison with the alternatives.

Override an existing keyword. This is not desirable for reasons including the stability of the language and flexibility.

New link attribute. This would add a new link attribute, making `sandboxed_lib` a new type next to the likes of `dylib` and `static`. Variations of this theme (like a new attribute type `protection="sandbox"`) would also be possible. This offers the highest level of abstraction for an approach directly implemented at the library boundary, but as we have seen in Section 3.5.1, wrapper functions are a necessity.

New Application Binary Interface (ABI) type. Replacing the (implicit) `extern "C"` ABI type would put focus on the language divide, but is misleading, because the ABI level is insufficient for placing a compartmentalization mechanism.

New function attribute. Extending existing functions attributes like `#[inline]` would be the best fit semantically.

New keyword. This carries a similar meaning to a new function attribute, but extending the narrow set of keywords is an intrusive change to the language and this option would break the prevalence of `extern` for introducing external resources.

Closure-like annotation. This suggests a semantic similarity to closures, but is a poor fit with traditional function syntax.

Current syntax. The current macro syntax is inferior to function attributes as the most fitting abstraction, which are already widely used in Rust. It has the advantages of making the sandboxing scope visually clear, and offering an indication of the mechanism used for function transformation. While not offering the best semantical abstraction,

Method	Example
Override an existing keyword	`extern { fn c_func(); }`
New link attribute	`#[link(name = "snappy", kind="sandboxed_lib")]` `extern { fn c_func(); }`
New ABI Type	`extern "sandboxedC" { fn c_func(); }`
New function attribute	`#[sandboxed]` `fn wrapper_func() {};`

New keyword	`sandboxed { fn c_func(); }`
	or
	`sandboxed fn wrapper_func(){};`

Closure-like annotation	`let retval = sandbox | | wrapper_func();`
Current syntax (macro)	`sandbox!{fn wrapper_func()};`

Table 5.1: Sandcrust language integration alternatives

the current syntax is still idiomatic for an instrumentation of existing Rust code. This is emphasized by the syntax of the *lazy_static* extension introduced in Section 4.3.1, which is depicted in Listing 4.3 on page 36. Implementing a new attribute is possible with Syntax extensions, so it may be possible in stable Rust with a future incarnation of Rust's macro system. We will discuss future directions of the macro interface in Chapter 6.

5.2 Library Interaction

To evaluate the forms of library interaction with the main program supported by Sandcrust, this section follows the rundown of possible program-library interactions by Wu et al. [100, pp. 863-864] and adopts the discussion to Sandcrust's wrapper function interface. Wu et. al. identify the following forms of interaction:

- By-value parameter passing and return
- By-reference parameter passing and return
- Global variable
- Function callback
- Long jump

By-Value and By-Reference Parameter Passing and Return. Sandcrust's ability to pass a data type rests on a native or derived implementation of the *Serialize* and *Deserialize* traits of the Serde framework described in Section 4.2. This prominently excludes *raw pointers* as parameters to the wrapper function. Given that the purpose of the wrapper function is to provide a safe Rust interface to an unsafe C function, this limitation is by design. The handling of C pointers in the wrapper, which is executed in

the sandboxed process, makes any resulting *tight interactions* transparent to the sandbox mechanism, so the full range of parameters and return values supported by Rust's FFI is supported by Sandcrust.

Global Variable. Sandcrust offers special support for *mutable foreign globals* - mutable global variables defined in the C library - via the `sandcrust_wrap_global!{}` macro. This is intended to facilitate the design of safe wrapper functions. The update mechanism exhibits the same constraints on supported types as the general function interface, but for use cases that do not require updates to the variable after initialization, it suffices to set the global variable to the desired value before initializing the stateful sandbox, effectively lifting any restrictions on global variables.

Function Callback. Function callbacks from a library function are supported within the sandboxed process, as demonstrated in Section 5.3.2. The current prototype implementation offers no support for callbacks to the main process. Chapter 6 outlines how this can be implemented in future work. Closures as a Rust-specific way to pass a function will likely not be supported by a future callback mechanism, which is no limitation for wrapper functions concerned with providing a safe interface around a legacy library, because the programming pattern is not supported by C.

Long Jump. The `setjmp()` / `longjmp()` mechanism provides "nonlocal gotos", i.e. it enables a function to jump back up the stack to a calling function. It causes undefined behavior when the calling function has already returned at the time of the `longjmp` and is prone to memory leaks due to the incomplete execution of the functions leading to the jump [134]. While Rust's FFI does not provide abstractions for the mechanism, it can be used manually. A Rust function wrapping a C library relying on long jumps is able to use the mechanism locally in the sandboxed process in the wrapper function and would be able to utilize the function callback mechanism, once implemented. The second case study in Section 5.3.2 shows how a Long Jump can be used within a wrapper function without impairing the security guarantees of the safe part of the program.

Apart from supporting function callbacks into the main process, Sandcrust supports all forms of library interactions necessary for providing a safe interface to unsafe C libraries.

5.3 Case Studies

5.3.1 Snappy FFI Example

Our first example is based on the FFI example in chapter 5.9 of the Rust Book [135]. It is shown in Listing 5.1, with unchanged parts left out for brevity. The highlighted lines designate additions or modifications of the original example. Lines 1, 2 and 5 import the Sandcrust crate. A missing wrapper function is added with lines 73-77, and the invocation is adopted in line 80. Annotating existing wrapper functions with the `sandbox!{}` macro accounts for the remaining highlighted lines.

Altogether, using Sandcrust adds 14 lines and changes 1, amounting to 18% of the original code base in this small example of 82 SLOC (by count of Cloc[1]).

```
1  #[macro_use]
2  extern crate sandcrust;
3  extern crate libc;
4
5  use sandcrust::*;
6  use libc::*;
7
8  #[link(name = "snappy")]
9  extern {

24 }
25
26 sandbox!{
27 pub fn validate_compressed_buffer(src: &[u8]) -> bool {
28     unsafe {
29         snappy_validate_compressed_buffer(src.as_ptr(), src.len() as
    ↪  size_t) == 0
30     }
31 }
32 }
33
34 sandbox!{
35 pub fn compress(src: &[u8]) -> Vec<u8> {

48 }
49 }
50
51 sandbox!{
52 pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {

70 }
71 }
72
73 sandbox!{
74 pub fn max_compressed_len(len: usize) -> usize {
75     unsafe { snappy_max_compressed_length(len as size_t) as usize}
76 }
77 }
78
79 fn main() {
80     let x = max_compressed_len(100);
81     println!("max compressed length of a 100 byte buffer: {}", x);
82 }
```

Listing 5.1: Rust FFI example with Sandcrust

[1] https://github.com/AlDanial/cloc

5.3.2 PNG File Decoding

The second example demonstrates a complex use case for Sandcrust. In line with Wu et al. [100], libpng[2] is used for its complex interface and impressive number of assigned CVE IDs. To demonstrate the use of the stateful sandbox for complex libraries, the example program uses global libpng data structures and provides multiple wrapper functions, instead of hiding the complexity of libpng in just one decoding interface. Listing 5.2 shows the function signatures implemented to decode a PNG file. The full example is shown in Listing A.1 on page 55ff. Except for the `read_file` function that reads the

```
fn read_file(path: &str) -> Vec<u8>;
fn png_init() -> Result<(), String>;
fn is_png(buf: &[u8]) -> bool;
extern "C" fn callback(callback_png_ptr: *mut png_struct,
                       buf_ptr: *mut u8,
                       count: png_size_t);
fn decode_png(png_image: &[u8]) -> Result<Vec<Vec<u8>>, String>;
```

Listing 5.2: Function signatures for PNG file decoding

PNG file into a buffer, all functions run in the sandboxed process set up by Sandcrust. The `png_init` function initializes the library, indicating failure to the main program with its `Result` return value. Next, `is_png` provides the main program with a way to check if the file is in fact a PNG image. The `callback` function is called by libpng to read the buffer, as it natively only supports reading a file from a file descriptor, which is not available in the current Sandboxing model. To propagate errors encountered during libpng decoding, `decode_png` uses the long jump mechanism, as displayed in Listing 5.3.

```
107 if 0 != setjmp(png_set_longjmp_fn(png_ptr, longjmp, jmp_buf_size)) {
108     return Err("read failed in libpng".to_owned());
109 }
```

Listing 5.3: Use of setjmp in decode_png

As explained in Section 5.2, using the `setjmp` / `longjmp` mechanism is highly unsafe. However, libpng relies on it for error handling. Using Sandcrust, it is possible to restrict all unsafe code to the sandboxed process. We will evaluate the performance towards the end of the next section.

5.4 Performance

The performance of Sandcrust was evaluated on a Lenovo Thinkpad X250 with an Intel Core i7-5600U CPU and 8 GiB of DDR3 memory running Rust 1.16 on Arch Linux with a 4.9 Linux kernel. To measure raw Sandcrust performance, the *Sandheap* crate provides

[2] `http://www.libpng.org/pub/png/libpng.html`

an interface according to the sandbox implementation requirements of Section 3.4.3, but with an empty function body.

Each test program in the *Sandstorm* test suite is instrumented via a macro that runs the specified code block for a number of warm up rounds, and then starts collecting time differences between wrapped calls to the `clock_gettime()` function of the C standard library, using the `CLOCK_MONOTONIC_RAW` clock option, for a number of profile runs. During the measurements, the average overhead between two consecutive calls to the time stamp function was measured to be between $83ns$ and $85ns$ and subtracted from each measurement. With almost all measurements in the microsecond range, the timer resolution is precise enough to allow for a meaningful distinction of measurement results. To get realistic, yet static data for byte vectors, a helper function repeatedly reads the contents of the *html* test case[3] from Snappy's repository up to the maximum transmitted size into memory.

At the end of each profile run, the data is written out as comma-separated values (CSV). The resulting data file is processed using the Numpy package[4] and plotted with Matplotlib [136]. Unless stated otherwise, all numbers discussed refer to the median of the measurement results.

5.4.1 Sandcrust Primitives Overhead

In this section, we discuss the overhead introduced by Sandcrust's primitives, shown in Figure 5.1. For a minimal example, we wrap the *abs(3)* [137] function of the C standard library. The box plot shows the following data: The line in the middle of each box shows the median, the symbols show the mean of the collected data. The boxes extend to the first and third quartiles, i.e. from 25% to 75% of the measured values. The whiskers show the lowest value still within 1.5 interquartile range (IQR) (the value difference between the first and the third quartile) of the first quartile, and the highest value within 1.5 IQR of the third quartile.

The *baseline* column shows a barely measurable duration of 0.001-0.002μs for the local execution of the wrapper function. In the next column, Sandcrust's single invocation takes 1166μs and 1192μs for the pipe and SHM-based implementations, respectively, a clear advantage in overhead compared to the *first invocation* in a stateful sandbox. This is because the first invocation includes the overhead of 1469μs (1412μs for SHM) seen in the *init* column, which reflects the more complex initialization work of `sandcrust_init()` and `sandcrust_init_with_shm_size()` functions, respectively.

The slightly higher time of 1506μs and 1493μs for the *first invocation* with automatic setup is only marginally bigger than the sum of the initialization overhead and the overhead for follow-up invocations in the stateful sandbox of 4.54μs and 4.37μs respectively. Its remaining overhead can be explained by paging / caching effects, because the measurements of the *first invocation* function necessarily take place in a freshly spawned process. The repeated *follow-up* invocations benefit more directly from a warm up phase of the benchmark program, which is not available to the *init* and *first invocation* measurements. This is also indicated by the higher quantiles in these measurements, compared

[3] `https://github.com/google/snappy/blob/master/testdata/html`
[4] `http://www.numpy.org/`

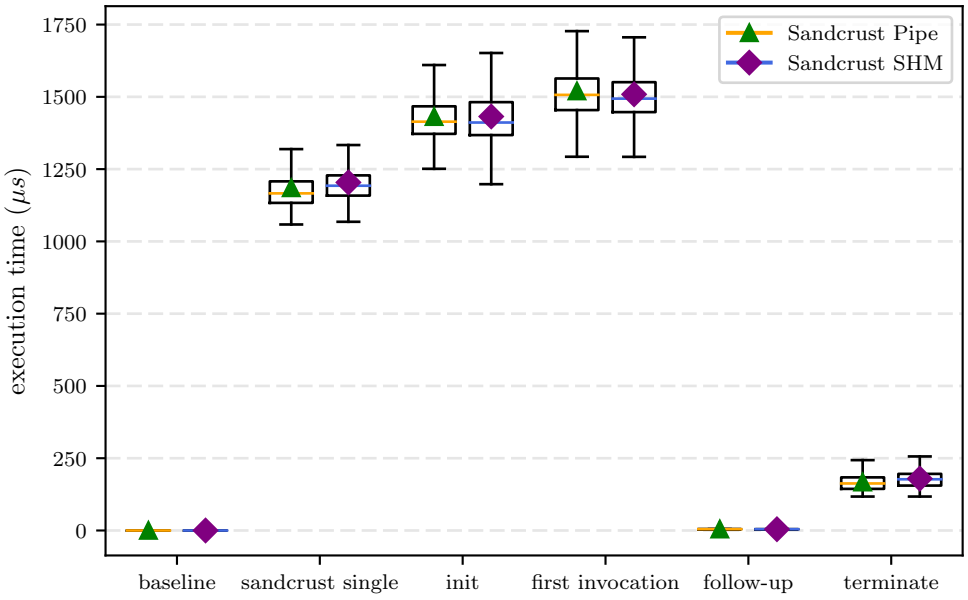


Figure 5.1: Sandcrust primitives overhead

to the narrow range in the rest of the measurements, where no warm up rounds were available. Lastly, explicit termination via `sandcrust_terminate()` takes around 162μs, or 177μs for the SHM version.

Differences between the pipe and the SHM versions are hardly visible. In place of a 4 byte buffered pipe write for data transmission in each direction, the SHM version issues an unbuffered pipe write from the sandboxed process to the parent to signal the readiness of the invocation results.

We will look into the IO effects of Sandcrust in more detail in the following section.

5.4.2 Sandcrust IO Overhead

One drawback of compartmentalized software is the overhead for data transfer and IPC. In the case of Sandcrust, additional overhead is introduced by the need to verify the data structures returned by the untrusted process. To quantify the effect of the different components, this test measures all possible combinations against a simple test case at data sizes from 4 bytes to 16 MiB.

The *Local (Memcopy)* function takes a reference to a Vector and a *slice* (a pointer to a window of a certain size) into a writable buffer as arguments. It then copies the contents of the vector into the buffer and returns another copy of the second buffer as a vector. For comparison, a second function (*Local (Bincode)*) serializes the Vector into the buffer, and returns a Vector as the result of deserializing the buffer.

The sandboxed function simply compares the first two bytes of a `&Vec<u8>` argument for equality. The vector is transmitted to the sandboxed process to update its state before the function invocation, which incurs the same copy into a buffer (or pipe) and return of a newly allocated vector copied out of the buffer. Therefore, albeit the functions are far from equal, this setup allows the direct comparison of the local and compartmentalized versions, because it requires the same amount of data transmissions and new memory allocations. Figure 5.2 yields surprising results:

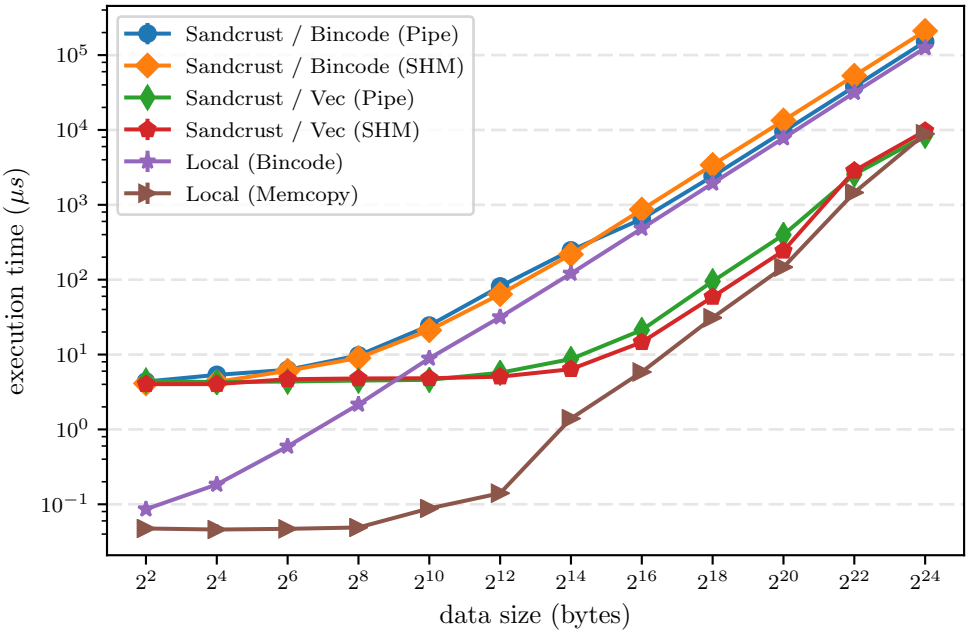


Figure 5.2: Comparison of IPC primitives

The *Local (Bincode)* graph shows a significant overhead over the *Local (Memcopy)* baseline. As shown in Figure 5.3, this overhead reaches a maximum at allocations of one page (4096 bytes), after which the relative slowdown goes down as the cost of copying data goes up. After an initial extra overhead of 4.35μs, the *Sandcrust / Bincode (Pipe)* overhead is quickly dominated by the Bincode overhead, with an additional slowdown of only 20-25% of the *Local (Bincode)* version.

The *Sandcrust / Bincode (SHM)* version exhibits a similar behavior, but surprisingly, the reduction of parallelism introduced by the sequence of "SHM write → Pipe ready signal send → Pipe ready signal receive → SHM read" reverses the slight speedup compared to *Sandcrust / Bincode (Pipe)* of 5-20% up to 16 KiB of transmitted data into a constant slowdown of 40% from 64 KiB upwards. This indicates a parallelism of serializing and deserializing which also explains the low overhead of the *Sandcrust / Bincode (Pipe)* variant compared to a local use of Bincode.

As the example uses byte vectors exclusively, it can benefit fully from Sandcrust's optional compile-time `custom_vec` optimization for byte field transmission. Indeed, the

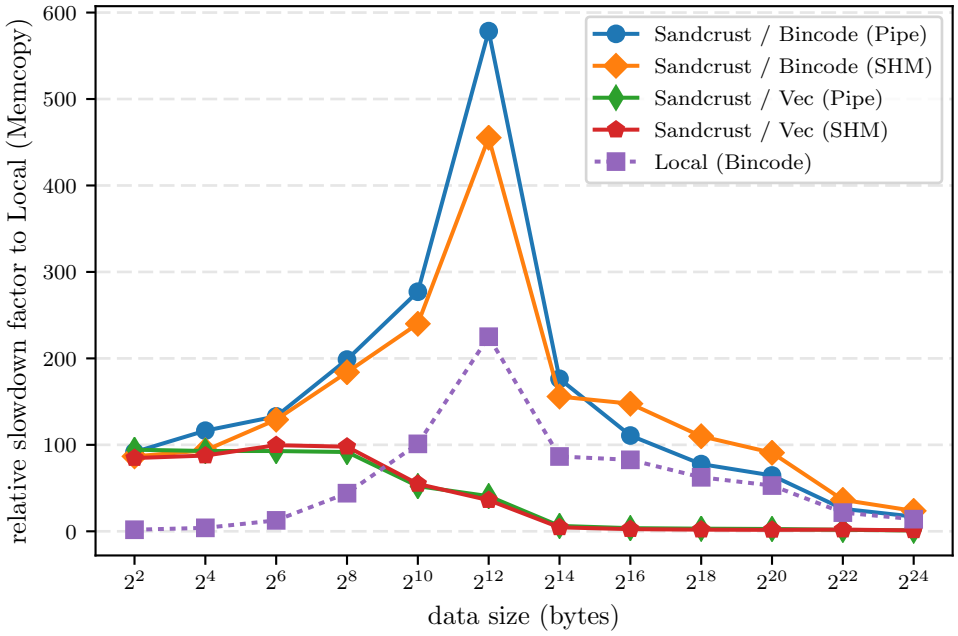


Figure 5.3: Relative slowdown of IPC primitives

overhead is reduced considerably and the graphs more closely follow the *Local (Memcopy)* baseline. Initially, both *Sandcrust / Vec* variants are on a par with each other. Starting from the test system's page size of 4096 or 2^{12} bytes, the *Sandcrust / Vec (SHM)* version shows a speedup of 11% of the pipe version. This can be explained by the pipe mechanism using more than one page for transmitting data, because the encoding adds an extra 8 bytes to the data size. The SHM version's advantage grows to the point of 39% at 2^{20} bytes or 1 MiB, after which the speedup rapidly turns into a slowdown as the system's cache size of 4 MiB is exceeded.

Altogether, we see from the measurements that an optimized alternative to Bincode would promise the highest potential for speeding up Sandcrust. Second, the use of SHM as a means of IPC has surprisingly little performance benefits compared to low-complexity pipe communication.

A comparison between IO primitives does not give a realistic assessment of a real-world use, because the biggest strength of monolithic software is that data can be passed between functions by reference and modified in place. As a comparison, Figure 5.4 shows the same measurements of Sandcrust's IPC options, but for a function that takes a Vector, modifies the first element and returns it. Strictly speaking, the `Vec` data type is passed by value, but it encapsulates a pointer to data on the heap.

The effect of sandboxing a function that only takes and returns the „(pointer, capacity, length) triplet“ [138] that comprises a Rust vector is that the data pointed to by the pointer has to be made available to the sandboxed process, and the modified result copied back in its entirety. This is of course disastrous for performance. Figure 5.4 shows

that for less than a page (2^{12}, or 4096 bytes) of data, the optimized Vector handling in Sandcrust incurs a moderate overhead of less than $5\mu s$, after which it exhibits a near-linear slowdown to the size of the vector, lower but similar to the Bincode version.

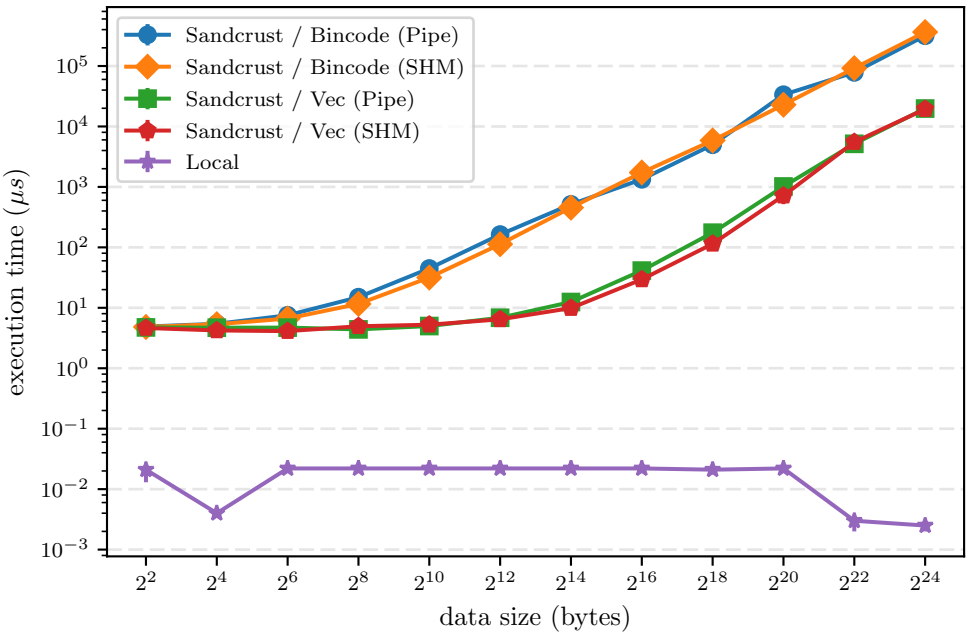


Figure 5.4: Worst-case copy overhead

5.4.3 Snappy Example Overhead

The extreme overhead of the last measurement is unrealistic in that the measured function has almost no execution time of its own, making the slowdown factor unrealistically high. The final measurement setup in this section is similar to the last section, but runs the `compress()` and `uncompress()` functions from Listing 5.1 for a select number of data sizes, using Sandcrust with the pipe transport and `custom_vec` optimization. Figure 5.5 shows the results.

There is a remarkable difference in overhead between the compress and the uncompress functions: The compress function goes down from a slowdown factor of 9.28 to a minimum of 1.3 at 2^{18} bytes, and stabilizes at a slowdown factor of around 1.5. The sandboxed uncompress function goes down steadily from a factor of 44 to 16.

This result can be explained by the wrapper signatures:

```
pub fn compress(src: &[u8]) -> Vec<u8>
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>>
```

The compress function's argument and result is handled entirely by the optimized transfer function for byte vectors. On the other hand, the uncompress result is wrapped in an `Option<T>` enum and thus handled by Bincode, which explains the increased slowdown.

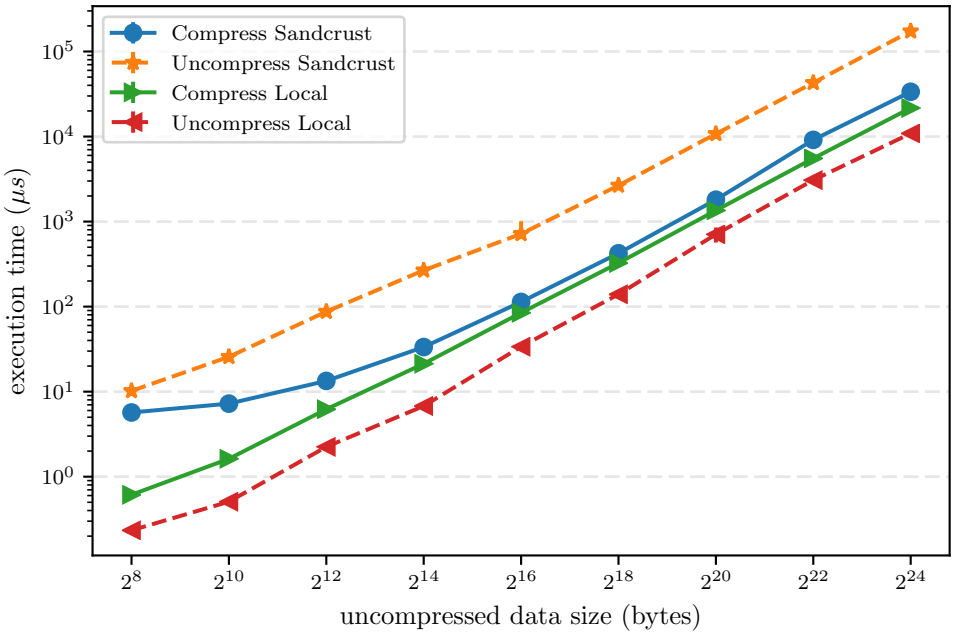


Figure 5.5: Snappy overhead

5.4.4 PNG File Decoding Overhead

To measure the effect of repeated calls to sandboxed functions, the last benchmark compares the PNG decoding example discussed in Section 5.3.2 against calling local functions. The test setup uses the pipe transport and `custom_vec` optimization. However, the effect of the latter is limited by the complex return value of `decode_png`, shown in Listing 5.2. The results are displayed in Figure 5.6, with the test file information shown in Table 5.2[5].

The performance measurements are consistent with the preceding tests: The absolute overhead from 2548μs for file 1 up to 26001μs for file 4 is considerably larger than comparable transfer sizes of the Snappy uncompress example, but well below the extreme worst-case copy overhead shown in Figure 5.4. Put in perspective, the relative slowdown factor shown for each test file in Table 5.2 compares favorably to the single-function overhead in Section 5.4.3. At a factor of 7.4, the PNG example starts out lower than the optimized transfers of the sandboxed compress function and reaches a minimum at file 3, similar in size to the sweet spot of 2^{18} bytes for the compress function. Altogether, while the absolute overheads of each sandboxed function call clearly add up, the relative

[5] The file URLs are:
`https://www.rust-lang.org/logos/rust-logo-256x256-blk.png`
`http://files.explosm.net/comics/Rob/myblood.png`
`https://colorblindprogramming.com/wp-content/uploads/2013/06/objects.png`
`http://natashenka.ca/wp-content/uploads/2014/01/arithmetic28x11.png`

ID	File	Size (bytes)	Sandcrust slowdown factor
1	rust-logo-256x256-blk.png	7460	7.40
2	myblood.png	42693	6.98
3	objects.png	386814	2.07
4	arithmetic28x11.png	894174	3.25

Table 5.2: PNG decoding test data

slowdown compared to a local series of function calls shows that Sandcrust does not inflict a disproportionate slowdown on complex use cases.

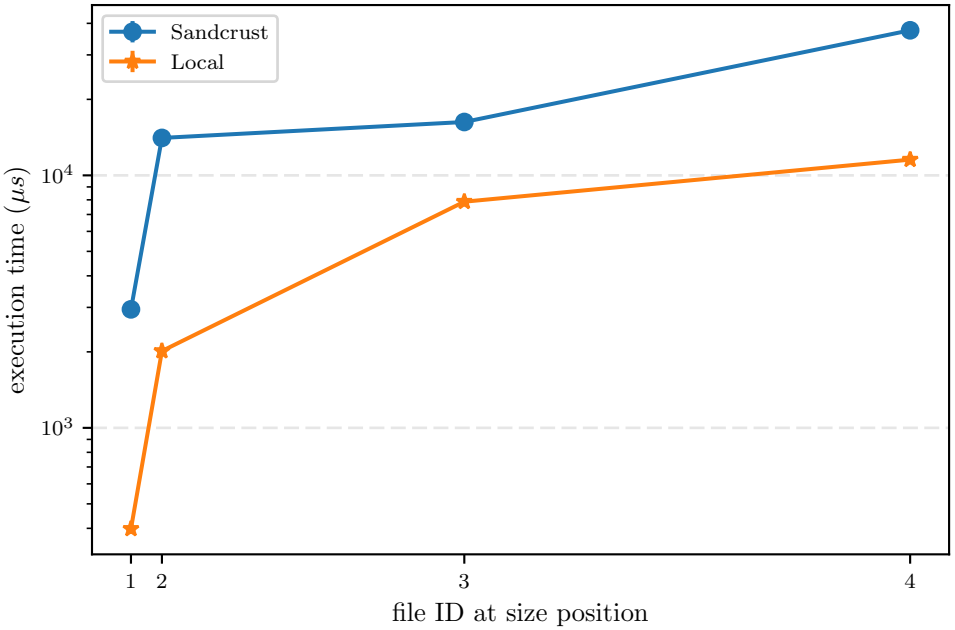


Figure 5.6: PNG file decoding overhead

The performance evaluation of Sandcrust shows that its overhead is dominated by the Bincode library. The overhead for short-running, low IO functions like library setup routines is large in relation to the original function run time, but in the realm of $5\mu s$ of absolute overhead, whereas for long-running functions, the runtime cost of IO would be negligible with an efficient data serialization and deserialization engine. With the current implementation, the slowdown factor is typically less than an order of magnitude and can be as little as 1.3, although due to Bincode's overhead, it can be substancially larger for IO-bound functions with complex parameter types. We will discuss future improvements in the next section.

6 Conclusion and Outlook

They always say that time changes things,
but you actually have to change them yourself.
— Andy Warhol

In the common practice of reusing unsafe software libraries inside otherwise safe Rust components, wrapper functions provide a safe interface around the unsafe component. By building a compartmentalization mechanism for those functions, *Sandcrust* is able to leverage Rust's strict typing system to automatically derive the information about data types that in prior work had to be provided manually via extensive annotations. The compartmentalization mechanism integrates seamlessly into a Rust project by utilizing Rust's macro system to transform the program. As a result, using Sandcrust is as simple as adding its crate and annotating each wrapper function to be sandboxed. This is in stark contrast to the external transformation tools, compiler and runtime modifications and addition of new programming language elements in prior work and the biggest contribution of this work.

From the decision to use macros follow a number of restrictions on the privilege separation model. The resulting requirement to sandbox an already running process is met by commodity sandboxing solutions like Linux' *Seccomp-BPF*. As discussed in Section 3.4.3, some use cases require careful handling of in-memory resources when deploying Sandcrust.

The prototype implementation lacks support for function callbacks into the main program, which is left for future work. This can be implemented by using the techniques developed for RPCs into a stateful sandbox in the reverse direction: Analogous to the function generation in Section 4.3.3, an appropriate RPC interface can be generated from an annotated callback function. In contrast to the endpoint implementation of Section 4.3.2, the generated RPC endpoint would provide a fixed interface to the annotated function instead of allowing to call arbitrary remote functions. Instead of the blocking pipe read currently performed after issuing an RPC, the main process would use an asynchronous IO mechanism[1] to concurrently accept an RPC from the sandboxed process. It may be necessary to provide a wrapper around the instrumented callback function that performs conversions of complex C data types before issuing the actual callback into the main process.

The performance evaluation has identified *Bincode*, the external Rust library used for (de-)serializing data for IPC, as having the largest impact on the overhead of Sandcrust. Speeding up Bincode is therefore the most tangible future work to improve on Sandcrust's performance. The remaining overhead measured for an optimized case without Bincode

[1] Unfortunately, Rust does not offer an interface to the *select()* system call yet: `https://github.com/rust-lang/rfcs/issues/1081`

can be attributed almost entirely to Linux IPC primitives used by Sandcrust. The overhead is therefore inherent to commodity OS's that lack highly optimized IPC mechanisms, as available in microkernels like L4 [35] or Nova [139].

Another area of improvement is the tooling available to users of Sandcrust. Once available in Rust, custom compile time errors[2] may provide better feedback if a user tries to sandbox a function with an incompatible signature. A port of LLVM-based SOAAP [140] to Rust would enable the user to test *compartmentalization hypotheses* for a program compartmentalized with Sandcrust, to validate the protection of sensitive data structures from unsafe code.

This work raises the question of how to best abstract the paradigm of componentization in a programming language. Declaring the use of Sandcrust for a function via a function attribute will likely be possible once *Token-based syntax extensions* [141] are part of Rust's stable interface. Although they would require a rewrite of Sandcrust's macro transformation system, the existing compartmentalization mechanism can be reused.

A remaining limitation is that all forms of Rust macros and syntax extensions only work on a subset of the program's token tree. This is the primary reason for Sandcrust's privilege separation model, as it limits any transformation of the existing program to the annotated functions. Hence, automatic transformation of the program's initialization is not possible from the annotated function. More far-reaching compiler plugins could eliminate Sandcrust's limitations on sandboxing library initialization routines and enable a different privilege separation model that clears memory resources by re-executing the process. A deeper access to compiler state could also enable sandboxing all functions originating from a specific crate or executing unsafe code. However, access to such far-ranging compiler intrinsics is unlikely to ever be offered in Rust's stable interface. With the proliferation of sandboxing mechanisms in commodity OS's, native language abstractions for sandboxed execution could eliminate most of the shortcomings without the need for complex compiler plugins.

This work shows that it is possible to make componentization an integral part of software development. To move beyond damage limitation in the face of unsafe monolithic legacy software towards least privilege components, safe programming languages should offer abstractions for the development of componentized software.

[2] `https://github.com/rust-lang/rust/issues/40872`

Appendix A

Source Code Listings

Listing A.1: Full PNG file decoding example

```
#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
extern crate libc;

use libc::{c_char, c_void, size_t, c_int};

#[macro_use]
extern crate sandcrust;
use sandcrust::*;

use std::fs::File;
use std::os::unix::fs::MetadataExt;
use std::ptr;
use std::io::Read;

type png_struct = c_void;
type png_info = c_void;
type png_size_t = size_t;

#[link(name = "png")]
extern "C" {
    fn png_create_read_struct(user_png_ver: *const c_char,
                              error_ptr: *mut c_void,
                              error_fn: *mut u8,
                              warn_fn: *mut u8)
                              -> *mut png_struct;
    fn png_create_info_struct(png_ptr: *mut png_struct)
                              -> *mut png_info;
    fn png_sig_cmp(sig: *const u8,
                   start: size_t,
                   num_to_check: size_t)
                   -> c_int;
    fn png_destroy_read_struct(png_ptr_ptr: *mut *mut png_struct,
                               info_ptr_ptr: *mut *mut png_info,
                               end_info_ptr_ptr: *mut *mut png_info);
    fn png_set_read_fn(png_ptr: *mut png_struct,
                       io_ptr: *mut c_void,
                       read_data_fn: extern "C" fn(*mut png_struct,
                                                   *mut u8,
                                                   size_t));
    fn png_get_io_ptr(png_ptr: *mut png_struct) -> *mut c_void;
```

```
42
43      fn png_read_info(png_ptr: *mut png_struct,
44                       info_ptr: *mut png_info);
45      fn png_get_image_height(png_ptr: *mut png_struct,
46                              info_ptr: *mut png_info)
47                              -> u32;
48      fn png_get_rowbytes(png_ptr: *mut png_struct,
49                          info_ptr: *mut png_info)
50                          -> u32;
51
52      fn png_read_image(png_ptr: *mut png_struct,
53                        row_pointers: *mut *mut u8);
54
55      fn setjmp(env: *mut c_void) -> c_int;
56      fn longjmp(env: *mut c_void, val: c_int);
57      fn png_set_longjmp_fn(png_ptr: *mut png_struct,
58                            longjmp_fn: unsafe extern "C" fn(*mut c_void,
59                                                             c_int),
60                            jmp_buf_size: size_t)
61  -> *mut c_void;
62  }
63
64
65  #[warn(non_camel_case_types)]
66  static mut png_ptr: *mut png_struct = 0 as *mut png_struct;
67  static mut info_ptr: *mut png_info = 0 as *mut png_struct;
68
69
70  /// Read file at path into a buffer.
71  fn read_file(path: &str) -> Vec<u8> {
72      let mut file = File::open(path).unwrap();
73      let size = file.metadata().unwrap().size() as usize;
74      let mut buf = vec!(0u8; size);
75      file.read_exact(&mut buf).unwrap();
76      buf
77  }
78
79
80  /// Custom read function for libpng.
81  extern "C" fn callback(callback_png_ptr: *mut png_struct,
82                         buf_ptr: *mut u8,
83                         count: png_size_t) {
84      unsafe {
85          let mut buf = std::slice::from_raw_parts_mut(buf_ptr,
86                                                       count as usize);
87          let image_ptr = png_get_io_ptr(callback_png_ptr);
88          let image: &mut &[u8] = ::std::mem::transmute(image_ptr);
89          image.read_exact(&mut buf).unwrap();
90      }
91  }
92
93
94  /// Read png image into a vector of row byte vectors.
95  sandbox!{
96  fn decode_png(png_image: &[u8]) -> Result<Vec<Vec<u8>>, String> {
```

```
97      unsafe {
98          // jmp buf size is 200 in this particular setup
99          let jmp_buf_size: size_t = 200;
100
101         // this call mimics the define in png.h:
102         // #  define png_jmpbuf(png_ptr) \
103         // (*png_set_longjmp_fn((png_ptr), longjmp, (sizeof (jmp_buf))))
104         if 0 != setjmp(png_set_longjmp_fn(png_ptr, longjmp,
    ↪  jmp_buf_size)) {
105             return Err("read failed in libpng".to_owned());
106         }
107
108         let image_ptr: *mut c_void = ::std::mem::transmute(&png_image);
109         png_set_read_fn(png_ptr, image_ptr, callback);
110
111         png_read_info(png_ptr, info_ptr);
112         let height = png_get_image_height(png_ptr, info_ptr) as usize;
113
114         let mut result = Vec::with_capacity(height);
115         let rowbytes = png_get_rowbytes(png_ptr,info_ptr) as usize;
116
117         // internal array of row pointers to feed to libpng
118         let mut rows = vec![ptr::null_mut() as *mut u8; height];
119
120         for i in 0..height {
121             let mut row = vec![0u8; rowbytes];
122             rows[i] = row.as_mut_ptr();
123             result.push(row);
124         }
125         png_read_image(png_ptr, rows.as_mut_ptr());
126         Ok(result)
127     }
128 }
129 }
130
131
132 /// Initialize libpng.
133 sandbox!{
134 fn png_init() -> Result<(), String> {
135     unsafe {
136         // for now, duplication is necessary
137         // https://stackoverflow.com/questions/21485655/how-do-i-use-c-pr
    ↪  eprocessor-macros-with-rusts-ffi
138         let ver = std::ffi::CString::new("1.6.28").unwrap();
139         let ver_ptr = ver.as_ptr();
140
141         png_ptr = png_create_read_struct(ver_ptr, ptr::null_mut(),
    ↪  ptr::null_mut(), ptr::null_mut());
142         if png_ptr.is_null() {
143             return Err("failed to create png_ptr".to_owned());
144         }
145         info_ptr = png_create_info_struct(png_ptr);
146         if info_ptr.is_null() {
147             png_destroy_read_struct(&mut png_ptr, ptr::null_mut(),
    ↪  ptr::null_mut());
```

```
148                 return Err("failed to create info_ptr".to_owned());
149             }
150         }
151         return Ok(());
152     }
153     }
154
155
156     /// Test argument file type.
157     sandbox!{
158     fn is_png(buf: &[u8]) -> bool {
159         let buf_ptr = buf.as_ptr();
160         let size = buf.len() as size_t;
161         unsafe {
162             if png_sig_cmp(buf_ptr, 0, size) != 0 {
163                 return false;
164             }
165         }
166         return true;
167     }
168     }
169
170
171     fn main() {
172         if let Some(arg1) = std::env::args().nth(1) {
173             let file_buf = read_file(&arg1.as_str());
174             if !is_png(&file_buf[0..8]) {
175                 panic!("no PNG!");
176             }
177             png_init().unwrap();
178             #[allow(unused_variables)]
179             let vec = decode_png(&file_buf).unwrap();
180         } else {
181             println!("usage: png <png file>");
182         }
183     }
```

Acronyms

ABI Application Binary Interface 41

ACL Access Control List 5, 6, 14

API application programming interface 5, 6, 13–15, 18, 19, 23, 25, 27, 29, 34

ASLR address space layout randomization 4

AST Abstract Syntax Tree 18–20, 31, 34, 37–39

BBN Bayesian Belief Network 3

CFI Control Flow Integrity 7

COM Component Object Model 8

CPU Central Processing Unit 16

CSV comma-separated values 46

CVE Common Vulnerabilities and Exposures 44

DIFC Decentralized Information Flow Control 8

DRTM dynamic root of trust for measurement 11

EPT Extended Page Table 11

FFI Foreign Function Interface 12, 13, 19, 25, 42, 43

IO Input / Output 17, 33, 47, 49, 51

IPC Inter-process communication 15, 17, 20, 22, 25, 26, 29, 31, 33, 35, 36, 38, 47, 49

IQR interquartile range 46

OS operating system 1, 2, 5–7, 20, 22

PDF Portable Data Format 7

PID Process ID 20

PNG Portable Network Graphics 45

RAII Resource Acquisition Is Initialization 12

RAM Random access memory 33

RPC Remote Procedure Call 8, 10, 22, 25, 26, 35–38

SFI Software Fault Isolation 7, 11

SHM shared memory 22, 29, 34, 46–49

SIP Software-Isolated Processes 6

SLOC Source Lines of Code 3, 10, 43

TCB Trusted Computing Base 4, 5, 7, 10, 11, 18, 20, 33

URL Uniform Resource Locator 51

vDSO virtual dynamic shared object 22

VMM Virtual Machine Monitor 9, 11

Bibliography

[1] *TIOBE Index | TIOBE - The Software Quality Company.* URL: http://www.tiobe.com/tiobe-index/ (visited on Mar. 3, 2017).

[2] *The 2016 Top Programming Languages - IEEE Spectrum.* URL: http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages (visited on Mar. 3, 2017).

[3] Raoul Strackx and Frank Piessens. „Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware“. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 2–13.

[4] Hermann Härtig et al. „DROPS: OS Support for Distributed Multimedia Applications“. In: *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. EW 8. Sintra, Portugal: ACM, 1998, pp. 203–209.

[5] *non-O(n) musings: Which projects should convert to Rust?* URL: http://jamey.thesharps.us/2017/01/which-projects-should-convert-to-rust.html (visited on Mar. 23, 2017).

[6] Algirdas Avizienis et al. „Basic Concepts and Taxonomy of Dependable and Secure Computing.“ In: *IEEE Trans. Dependable Sec. Comput.* 1.1 (2004), pp. 11–33.

[7] Myron Lipow. „Number of faults per line of code“. In: *IEEE Transactions on Software Engineering* 4 (1982), pp. 437–439.

[8] Thomas J. Ostrand and Elaine J. Weyuker. „The Distribution of Faults in a Large Industrial Software System“. In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '02. Roma, Italy: ACM, 2002, pp. 55–64.

[9] Fumio Akiyama. „An Example of Software System Debugging“. In: *IFIP Congress (1)*. 1971, pp. 353–359.

[10] Arthur E. Ferdinand. „A theory of system complexity“. In: *International Journal of General System* 1.1 (1974), pp. 19–33.

[11] Norman E. Fenton and Martin Neil. „A Critique of Software Defect Prediction Models.“ In: *IEEE Trans. Software Eng.* 25.5 (1999), pp. 675–689.

[12] Norman E Fenton and Martin Neil. „Software metrics: successes, failures and new directions“. In: *Journal of Systems and Software* 47.2 (1999), pp. 149–157.

[13] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. „Mining metrics to predict component failures“. In: *Proc. Int'l Conf. Software Engineering (ICSE)*. Shanghai, China: ACM Press, 2006, pp. 452–461.

[14] Trevor Jim et al. „Cyclone: A safe dialect of C“. In: *Proceedings of the 2002 USENIX Annual Technical Conference*. 2002.

[15] George Necula, Scott McPeak, and Westley Weimer. „CCured: Type-Safe Retrofitting of Legacy Code“. In: *Proc. Principles of Programming Languages (POPL'02)*. ACM, 2002.

[16] Aleph One. *Smashing the stack for fun and profit. Phrack, 7 (49), November 1996.*

[17] Yves Younan, Wouter Joosen, and Frank Piessens. *Code injection in C and C++ : A survey of vulnerabilities and countermeasures*. Tech. rep. Departement Computerwetenschappen, Katholieke Universiteit Leuven, 2004.

[18] Úlfar Erlingsson, Yves Younan, and Frank Piessens. „Low-Level Software Security by Example“. In: *Handbook of Information and Communication Security*. Springer, 2010.

[19] Raoul Strackx et al. „Breaking the memory secrecy assumption.“ In: *EUROSEC*. Ed. by Evangelos P. Markatos and Manuel Costa. ACM, 2009, pp. 1–8.

[20] Ben Gras et al. „ASLR on the Line: Practical Cache Attacks on the MMU“. In: *NDSS*. Feb. 2017.

[21] Hovav Shacham. „The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).“ In: *ACM Conference on Computer and Communications Security*. Ed. by Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson. ACM, 2007, pp. 552–561.

[22] Kayvan Memarian et al. „Into the depths of C: elaborating the de facto standards“. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2016, pp. 1–15.

[23] J. H. Saltzer and M. D. Schroeder. „The protection of information in computer systems“. In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308. ISSN: 0018-9219.

[24] John Rushby. „Design and Verification of Secure Systems“. In: *Reprint of a paper presented at the 8th ACM Symposium on Operating System Principles*. Vol. 15. 5. Pacific Grove, California, Dec. 1981, pp. 12–21.

[25] Lili Qiu et al. „Trusted Computer System Evaluation Criteria“. In: *National Computer Security Center*. 1985.

[26] Robert Wahbe et al. „Efficient Software-Based Fault Isolation.“ In: *SOSP*. Ed. by Andrew P. Black and Barbara Liskov. Operating System Review 27(5). ACM, 1993, pp. 203–216.

[27] Ian Goldberg et al. „A Secure Environment for Untrusted Helper Applications.“ In: *USENIX Security*. USENIX Association, 1996.

[28] Jack B. Dennis and Earl C. Van Horn. „Programming Semantics for Multiprogrammed Computations“. In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155. ISSN: 0001-0782.

[29] Jerome H Saltzer and Michael D Schroeder. „The protection of information in computer systems“. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

[30] M. D. McIlroy, E. N. Pinson, and B. A. Tague. „Unix Time-Sharing System Forward“. In: *The Bell System Technical Journal* 57.6, part 2 (1978), p.1902.

[31] Peter G Neumann et al. „A provably secure operating system: The system, its applications, and proofs“. In: *Computer Science Laboratory Report CSL-116,* (1980).

[32] Jonathan Woodruff et al. „The CHERI capability model: Revisiting RISC in an age of risk“. In: *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE. 2014, pp. 457–468.

[33] Ellis Cohen and David Jefferson. „Protection in the Hydra operating system“. In: *ACM SIGOPS Operating Systems Review*. Vol. 9. 5. ACM. 1975, pp. 141–160.

[34] Michael J. Accetta et al. „Mach: A New Kernel Foundation for UNIX Development.“ In: *USENIX Summer*. USENIX Association, 1986, pp. 93–113.

[35] J. Liedtke. „On Micro-kernel Construction“. In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 237–250.

[36] Adam Lackorzynski and Alexander Warg. „Taming Subsystems: Capabilities As Universal Resource Access Control in L4“. In: *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*. IIES '09. Nuremburg, Germany: ACM, 2009, pp. 25–30.

[37] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. „EROS: a fast capability system.“ In: *SOSP*. Ed. by David Kotz and John Wilkes. Operating System Review 33(5). ACM, 1999, pp. 170–185.

[38] Arindam Banerji, John M Tracey, and David L Cohn. „Protected Shared Libraries—a new approach to modularity and sharing“. In: *Proceedings of the 1997 USENIX Technical Conference*. 1997, pp. 59–75.

[39] Jeffrey S. Chase et al. „Sharing and Protection in a Single-Address-Space Operating System.“ In: *ACM Trans. Comput. Syst.* 12.4 (1994), pp. 271–307.

[40] Galen Hunt et al. „Sealing OS processes to improve dependability and safety“. In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 3. ACM. 2007, pp. 341–354.

[41] Galen C Hunt and James R Larus. „Singularity: rethinking the software stack“. In: *ACM SIGOPS Operating Systems Review* 41.2 (2007), pp. 37–49.

[42] Christian Jensen and Daniel Hagimont. „Protection Wrappers: A Simple and Portable Sandbox for Untrusted Applications“. In: *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. EW 8. Sintra, Portugal: ACM, 1998, pp. 104–110.

[43] Kapil Jain and R Sekar. „User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement.“ In: *NDSS*. 2000.

[44] Poul-Henning Kamp and Robert NM Watson. „Jails: Confining the omnipotent root“. In: *Proceedings of the 2nd International SANE Conference*. Vol. 43. 2000, p. 116.

[45] Niels Provos. „Improving Host Security with System Call Policies.“ In: *Usenix Security*. Vol. 3. 2003, p. 19.

[46] Ma Bo et al. „Improvements the Seccomp sandbox based on PBE theory“. In: *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*. IEEE. 2013, pp. 323–328.

[47] NSA Peter Loscocco. „Integrating flexible support for security policies into the Linux operating system“. In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference. Boston: USENIX Association*. 2001.

[48] Robert NM Watson et al. „A taste of Capsicum: practical capabilities for UNIX“. In: *Communications of the ACM* 55.3 (2012), pp. 97–104.

[49] Takahiro Shinagawa, Kenji Kono, and Takashi Masuda. „Flexible and efficient sandboxing based on fine-grained protection domains“. In: *Software Security—Theories and Systems*. Springer, 2003, pp. 172–184.

[50] *System and kernel security | Android Open Source Project*. URL: `https://source.android.com/security/overview/kernel-security.html#the-application-sandbox` (visited on Feb. 21, 2017).

[51] *Linux Sandboxing*. URL: `https://chromium.googlesource.com/chromium/src/+/master/docs/linux_sandboxing.md` (visited on Feb. 21, 2017).

[52] Taesoo Kim and Nickolai Zeldovich. „Practical and Effective Sandboxing for Non-root Users.“ In: *USENIX Annual Technical Conference*. 2013, pp. 139–144.

[53] Stephen McCamant and Greg Morrisett. „Efficient, verifiable binary sandboxing for a CISC architecture“. In: (2005).

[54] Stephen McCamant and Greg Morrisett. „Evaluating SFI for a CISC Architecture.“ In: *Usenix Security*. Vol. 6. 2006.

[55] Bryan Ford and Russ Cox. „Vx32: Lightweight User-level Sandboxing on the x86.“ In: *USENIX Annual Technical Conference*. Ed. by Rebecca Isaacs and Yuanyuan Zhou. USENIX Association, 2008, pp. 293–306.

[56] Úlfar Erlingsson et al. „XFI: Software Guards for System Address Spaces“. In: *OSDI*. Ed. by Brian N. Bershad and Jeffrey C. Mogul. USENIX Association, 2006, pp. 75–88.

[57] Bennet Yee et al. „Native Client: A Sandbox for Portable, Untrusted x86 Native Code“. In: *IEEE Symposium on Security and Privacy (Oakland'09)*. New York, NY, USA, 2009.

[58] Greg Morrisett et al. „RockSalt: better, faster, stronger SFI for the x86.“ In: *PLDI*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 395–404.

[59] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. „Improving the reliability of commodity operating systems.“ In: *SOSP*. Ed. by Michael L. Scott and Larry L. Peterson. ACM, 2003, pp. 207–222.

[60] Sotiris Ioannidis and Steven M Bellovin. „Building a Secure Web Browser.“ In: *USENIX Annual Technical Conference, FREENIX Track*. 2001, pp. 127–134.

[61] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan M. Smith. „Sub-operating Systems: A New Approach to Application Security“. In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. Saint-Emilion, France: ACM, 2002, pp. 108–115.

[62] Matt Piotrowski and Anthony D Joseph. *Virtics: A system for privilege separation of legacy desktop applications*. Tech. rep. Technical Report UCB/EECS-2010-70, UC Berkeley, 2010.

[63] David S Peterson, Matt Bishop, and Raju Pandey. „A Flexible Containment Mechanism for Executing Untrusted Code.“ In: *Usenix Security Symposium*. 2002, pp. 207–225.

[64] Michael Maass et al. „A systematic analysis of the science of sandboxing“. In: *PeerJ Computer Science* 2 (2016), e43.

[65] A. van Dam, G. M. Stabler, and R. J. Harrington. „Intelligent Satellites for Interactive Graphics“. In: *Proceedings of the IEEE* 62.4 (Apr. 1974), pp. 483–492. ISSN: 0018-9219.

[66] Griffith Hamlin Jr. and James D. Foley. „Configurable Applications for Graphics Employing Satellites (CAGES)“. In: *SIGGRAPH Comput. Graph.* 9.1 (Apr. 1975), pp. 9–19. ISSN: 0097-8930.

[67] Galen C. Hunt and Michael L. Scott. „The Coign Automatic Distributed Partitioning System“. In: *OSDI*. Ed. by Margo I. Seltzer and Paul J. Leach. USENIX Association, 1999, pp. 187–200.

[68] Steve Zdancewic et al. „Secure program partitioning.“ In: *ACM Trans. Comput. Syst.* 20.3 (2002), pp. 283–328.

[69] Stephen Chong et al. „Secure Web Applications via Automatic Partitioning“. In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 31–44.

[70] Andrew C. Myers and Barbara Liskov. „A Decentralized Model for Information Flow Control.“ In: *SOSP*. Ed. by Michel Banâtre, Henry M. Levy, and William M. Waite. Operating System Review 31(5). ACM, 1997, pp. 129–142.

[71] Andrew C Myers. „JFlow: Practical mostly-static information flow control“. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1999, pp. 228–241.

[72] Andrew C. Myers and Barbara Liskov. „Protecting privacy using the decentralized label model.“ In: *ACM Trans. Softw. Eng. Methodol.* 9.4 (2000), pp. 410–442.

[73] Dan Søndergaard et al. „Program Partitioning Using Dynamic Trust Models“. In: *Formal Aspects in Security and Trust*. Springer, 2007.

[74] Lantian Zheng et al. „Using Replication and Partitioning to Build Secure Distributed Systems.“ In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2003, pp. 236–250.

[75] Aaron Blankstein and Michael J. Freedman. „Automating Isolation and Least Privilege in Web Services“. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 133–148.

[76] Maxwell Krohn et al. „Information Flow Control for Standard OS Abstractions“. In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 321–334.

[77] Andrea Bittau et al. „Wedge: Splitting Applications into Reduced-privilege Compartments“. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'08. San Francisco, California: USENIX Association, 2008, pp. 309–322.

[78] Jun Wang, Xi Xiong, and Peng Liu. „Practical Fine-grained Privilege Separation in Multithreaded Applications“. In: *CoRR* abs/1305.2553 (2013).

[79] Nickolai Zeldovich et al. „Making information flow explicit in HiStar.“ In: *Commun. ACM* 54.11 (2011), pp. 93–101.

[80] Karl Marx. „Das Kapital–Kritik der politischen Ökonomie (Capital–A Critique of Political Economy)“. In: *Hamburg, Germany: Otto Meissner* (1867).

[81] Niels Provos, Markus Friedl, and Peter Honeyman. „Preventing Privilege Escalation“. In: *12th USENIX Security Symposium* (Aug. 2003), p. 11.

[82] Douglas Kilpatrick. „Privman: A Library for Partitioning Applications.“ In: *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 2003, pp. 273–284.

[83] David Brumley and Dawn Song. „Privtrans: Automatically Partitioning Programs for Privilege Separation“. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004.

[84] Y. Wu et al. „Automatically partition software into least privilege components using dynamic data dependency analysis“. In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. Nov. 2013, pp. 323–333.

[85] Hermann Härtig et al. „The Nizza secure-system architecture“. In: *CollaborateCom*. 2005.

[86] Lenin Singaravelu et al. „Reducing TCB complexity for security-sensitive applications: Three case studies“. In: *ACM SIGOPS Operating Systems Review*. Vol. 40. 4. ACM. 2006, pp. 161–174.

[87] Carsten Weinhold and Hermann Härtig. „VPFS: Building a Virtual Private File System with a Small Trusted Computing Base“. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. Eurosys '08. Glasgow, Scotland UK: ACM, 2008, pp. 81–93.

[88] Carsten Weinhold and Hermann Härtig. „jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components“. In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'11. Portland, OR, USA: USENIX Association, 2011, pp. 32–32.

[89] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. „Improving Xen Security through Disaggregation“. In: *VEE*. Ed. by David Gregg, Vikram S. Adve, and Brian N. Bershad. ACM, 2008, pp. 151–160.

[90] Hermann Härtig et al. „The Performance of µKernel-Based Systems.“ In: *SOSP*. 1997, pp. 66–77.

[91] Richard Ta-Min, Lionel Litty, and David Lie. „Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable“. In: *OSDI*. Ed. by Brian N. Bershad and Jeffrey C. Mogul. USENIX Association, 2006, pp. 279–292.

[92] Derek Gordon Murray and Steven Hand. „Privilege separation made easy: trusting small libraries not big processes.“ In: *EUROSEC*. Ed. by Herbert Bos and Evangelos P. Markatos. ACM, 2008, pp. 40–46.

[93] Mingyuan Xia et al. „Enhanced privilege separation for commodity software on virtualized platform“. In: *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. IEEE. 2010, pp. 275–282.

[94] Jonathan M. McCune et al. „TrustVisor: Efficient TCB Reduction and Attestation.“ In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010, pp. 143–158.

[95] Yutao Liu et al. „Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation.“ In: *ACM Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM, 2015, pp. 1607–1619.

[96] Charles Reis and Steven D. Gribble. „Isolating Web Programs in Modern Browser Architectures“. In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 219–232.

[97] Ben Niu and Gang Tan. „Enforcing user-space privilege separation with declarative architectures“. In: *Proceedings of the seventh ACM workshop on Scalable trusted computing*. ACM. 2012, pp. 9–20.

[98] Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. „Quarantine: Fault Tolerance for Concurrent Servers with Data-Driven Selective Isolation“. In: *HotPar*. Ed. by Michael McCool and Mendel Rosenblum. USENIX Association, 2011.

[99] Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. „Fractured processes: adaptive, fine-grained process abstractions“. In: *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*. 2014.

[100] Yongzheng Wu et al. „Codejail: Application-Transparent Isolation of Libraries with Tight Program Interactions“. In: *Computer Security – ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*. Ed. by Sara Foresti, Moti Yung, and Fabio Martinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 859–876.

[101] *Variable Bindings*. URL: `https://doc.rust-lang.org/book/variable-bindings.html` (visited on Feb. 10, 2017).

[102] *Lifetimes*. URL: `https://doc.rust-lang.org/book/lifetimes.html` (visited on Feb. 10, 2017).

[103] *Ownership*. URL: `https://doc.rust-lang.org/book/ownership.html` (visited on Feb. 10, 2017).

[104] *References and Borrowing*. URL: `https://doc.rust-lang.org/book/references-and-borrowing.html` (visited on Feb. 10, 2017).

[105] *Concurrency*. URL: `file:///home/ben/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/share/doc/rust/html/book/concurrency.html` (visited on Mar. 3, 2017).

[106] *The Rust Reference*. URL: `https://doc.rust-lang.org/reference.html#memory-model` (visited on Feb. 9, 2017).

[107] *Unsafe*. URL: `https://doc.rust-lang.org/book/unsafe.html` (visited on Feb. 10, 2017).

[108] *The Rust Reference*. URL: `https://doc.rust-lang.org/reference.html#external-blocks` (visited on Feb. 10, 2017).

[109] *Namespaces in operation, part 1: namespaces overview [LWN.net]*. URL: `https://lwn.net/Articles/531114/` (visited on Feb. 9, 2017).

[110] *namespaces(7) - Linux manual page*. URL: `http://man7.org/linux/man-pages/man7/namespaces.7.html` (visited on Feb. 10, 2017).

[111] *Anatomy of a user namespaces vulnerability [LWN.net]*. URL: `https://lwn.net/Articles/543273/` (visited on Feb. 23, 2017).

[112] *seccomp(2) - Linux manual page*. URL: `http://man7.org/linux/man-pages/man2/seccomp.2.html` (visited on Feb. 23, 2017).

[113] YongBin Zhou and DengGuo Feng. „Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing.“ In: *IACR Cryptology ePrint Archive* 2005 (2005), p. 388.

[114] Michael Schwarz et al. „Malware Guard Extension: Using SGX to Conceal Cache Attacks“. In: *CoRR* abs/1702.08719 (2017).

[115] *syscalls(2) - Linux manual page*. URL: `http://man7.org/linux/man-pages/man2/syscalls.2.html` (visited on Feb. 26, 2017).

[116] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. „We Crashed, Now What?“ In: *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*. HotDep'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–8.

[117] *Macros*. URL: `https://doc.rust-lang.org/book/macros.html` (visited on Feb. 23, 2017).

[118] *Compiler Plugins*. URL: `https://doc.rust-lang.org/book/compiler-plugins.html` (visited on Feb. 23, 2017).

[119] *Crates and Modules*. URL: `https://doc.rust-lang.org/book/crates-and-modules.html` (visited on Feb. 23, 2017).

[120] *Stability as a Deliverable - The Rust Programming Language Blog*. URL: `https://blog.rust-lang.org/2014/10/30/Stability.html` (visited on Feb. 23, 2017).

[121] *Bash on Ubuntu on Windows - About*. URL: `https://msdn.microsoft.com/de-de/commandline/wsl/about` (visited on Mar. 9, 2017).

[122] *fork(2) - Linux manual page*. URL: `http://man7.org/linux/man-pages/man2/fork.2.html` (visited on Feb. 24, 2017).

[123] *wait(2) - Linux manual page*. URL: `http://man7.org/linux/man-pages/man2/wait.2.html` (visited on Feb. 24, 2017).

[124] *execve(2) - Linux manual page*. URL: `http://man7.org/linux/man-pages/man2/execve.2.html` (visited on Feb. 24, 2017).

[125] *vdso(7) - Linux manual page*. URL: `http://man7.org/linux/man-pages/man7/vdso.7.html` (visited on Mar. 30, 2017).

[126] U. Shankar and D. Wagner. „Preventing Secret Leakage from fork(): Securing Privilege-Separated Applications“. In: *2006 IEEE International Conference on Communications*. Vol. 5. June 2006, pp. 2268–2275.

[127] *clearenv(3) - Linux manual page*. URL: `http://man7.org/linux/man-pages/man3/clearenv.3.html` (visited on Feb. 26, 2017).

[128] Alexis Beingessner. *The Many Kinds of Code Reuse in Rust*. URL: `http://cglab.ca/~abeinges/blah/rust-reuse-and-recycle/` (visited on Mar. 9, 2017).

[129] *The Little Book of Rust Macros*. URL: `https://danielkeep.github.io/tlborm/book/README.html` (visited on Mar. 28, 2017).

[130] Bertrand Meyer and Karine Arnout. „Componentization: The Visitor Example.“ In: *IEEE Computer* 39.7 (2006), pp. 23–30.

[131] *Procedural Macros (and custom Derive)*. URL: `https://doc.rust-lang.org/book/procedural-macros.html` (visited on Mar. 9, 2017).

[132] *const and static*. URL: `https://doc.rust-lang.org/book/const-and-static.html` (visited on Mar. 18, 2017).

[133] *Push-down Accumulation*. URL: `https://danielkeep.github.io/tlborm/book/pat-push-down-accumulation.html` (visited on Mar. 18, 2017).

[134] *setjmp(3) - Linux manual page*. URL: http://man7.org/linux/man-pages/man3/longjmp.3.html (visited on Apr. 1, 2017).

[135] *Foreign Function Interface*. URL: https://doc.rust-lang.org/book/ffi.html (visited on Apr. 13, 2017).

[136] J. D. Hunter. „Matplotlib: A 2D graphics environment“. In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95.

[137] *abs(3) - Linux manual page*. URL: http://man7.org/linux/man-pages/man3/abs.3.html (visited on Apr. 14, 2017).

[138] *std::vec::Vec - Rust*. URL: https://doc.rust-lang.org/std/vec/struct.Vec.html (visited on Apr. 15, 2017).

[139] Udo Steinberg and Bernhard Kauer. „NOVA: A Microhypervisor-based Secure Virtualization Architecture“. In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 209–222.

[140] Khilan Gudka et al. „Clean Application Compartmentalization with SOAAP.“ In: *ACM Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM, 2015, pp. 1016–1031.

[141] *Macros (and syntax extensions and compiler plugins) - where are we at? - announcements - The Rust Programming Language Forum*. URL: https://users.rust-lang.org/t/macros-and-syntax-extensions-and-compiler-plugins-where-are-we-at/7600 (visited on Feb. 24, 2017).