TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code

Inyoung Bang and Martin Kayondo, *Seoul National University;* Hyungon Moon, *UNIST* (*Ulsan National Institute of Science and Technology*)*;* Yunheung Paek, *Seoul National University*

https://www.usenix.org/conference/usenixsecurity23/presentation/bang

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

TRUST: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code

Inyoung Bang[1,2] *Martin Kayondo*[1,2] *Hyungon Moon*[3,*]
Yunheung Paek[1,2,*]

[1]ECE, Seoul National University, [2]ISRC, Seoul National University
[3]UNIST

{iybang, kymartin}@sor.snu.ac.kr,
ypaek@snu.ac.kr, hyungon@unist.ac.kr

Abstract

Rust was invented to help developers build highly safe systems. It comes with a variety of programming constructs that put emphasis on safety and control of memory layout. Rust enforces strict discipline about a type system and ownership model to enable compile-time checks of all spatial and temporal safety errors. Despite this advantage in security, the restrictions imposed by Rust's type system make it difficult or inefficient to express certain designs or computations. To ease or simplify their programming, developers thus often include *untrusted code* from unsafe Rust or external libraries written in other languages. Sadly, the programming practices embracing such untrusted code for flexibility or efficiency subvert the strong safety guarantees by *safe* Rust. This paper presents TRUST, a compilation framework which against untrusted code present in the program, provides trustworthy protection of safe Rust via in-process isolation. Its main strategy is allocating objects in an isolated memory region that is accessible to safe Rust but restricted from being written by the untrusted. To enforce this, TRUST employs software fault isolation and x86 protection keys. It can be applied directly to any Rust code without requiring manual changes. Our experiments reveal that TRUST is effective and efficient, incurring runtime overhead of only 7.55% and memory overhead of 13.30% on average when running 11 widely used crates in Rust.

1 Introduction

C/C++ have been dominant languages for several decades, but they are unsafe due to the permissive semantics that allows many undefined behaviors often manifested as a myriad of security-critical pervasive bugs, such as buffer overflows, use-after-free, and other memory exploitable errors [17, 35, 41, 52, 54]. The language *Rust* [32, 45] was invented to tackle this security problem inherent in C/C++ by introducing new syntax and semantics such as pointer ownership, lifetime, and borrowing. The language features and constructs of Rust are elaborated to facilitate static analysis for safety guarantee, thereby requiring little or no runtime sanity checking. However, despite Rust's strong static security guarantees, its strict semantics and rules for the guarantees potentially limit expressiveness [25] as well as performance [50]. Thus for practicality, Rust relaxes its strict safety rules by allowing programmers to include *untrusted code*, which can be exempt from strict sanity checks at compile time. One source of the untrusted code is code sections in Rust, named *unsafe blocks*, which typically contain a handful of the operations crucial for low-level coding or critical for performance, such as raw pointer manipulation and unprotected type casting. Those operations wrapped within a Rust unsafe block are written in a second language beside Rust, called *unsafe Rust*, which does not necessarily obey all safety rules of the true Rust (preferably, *safe Rust*) language. As the other source of untrusted code, external libraries written in unknown languages are used by programmers to save their development costs. The programming practices encompassing such untrusted code with safe Rust indeed bring diverse benefits in terms of expressive power and efficiency, but will undermine the strong safety guarantees by safe Rust evidently because running untrusted code in the same address space as safe Rust code would put the entire program into danger of being exposed to exploits from unsafe blocks or external libraries.

In this paper, we aim to shed light on this potential security risk of Rust programming with untrusted code sources, and to propose our compilation framework, called TRUST, which is designed to mitigate the risk via *in-process isolation* where the Rust code is transformed by TRUST to quarantine unsafe blocks and external library functions from safe Rust code sections which we will say hereafter as *safe blocks*. We opted for an in-process scheme because it is generally deemed more efficient than the inter-process counterpart for isolation [28] that requires costly OS intervention for virtualization support. In fact, there are several in-process schemes [15, 31] previously developed to address this risk with such efficiency, but they have several limitations, such as lacking Rust-aware program analysis, requiring expensive context switches, or

*Corresponding authors

relying on developer annotations.

By quarantining unsafe blocks and external libraries from safe blocks, TRUST prevents exploits in such untrusted code from corrupting critical data of safe Rust without proper permissions. To implement our in-process isolation for Rust applications, TRUST first divides the memory into two regions, *safe* and *unsafe*, which contain safe objects and unsafe ones, respectively. Then, TRUST differentiates access permissions of Rust code blocks to these regions. It permits safe blocks to access both the regions. In contrast, it denies by default any access of unsafe blocks and external libraries to the safe region, although some untrusted code blocks may be given limited permission to read the safe region depending on its security policy. It is noteworthy here that TRUST can perform access control on every individual object in a Rust program by determining which objects are allocated to either of the regions. That is to say, as for safe stack/heap objects whose confidentiality and integrity are of top priority, TRUST allocates them into the safe region so that any access of untrusted code to them can be restricted, with the exception of some untrusted blocks receiving read permission to safe objects for special cases. All other objects will be classified as unsafe ones and placed into the unsafe region.

For automatic instrumentation of the Rust code for in-process isolation, we have made modifications to the Rust framework, including the frontend for IR generation and the backend for binary generation. When the Rust code is instrumented for in-process isolation, TRUST applies different techniques to cope with an unsafe block and an external library because the former can be presented to TRUST as source code, while the latter is assumed available only in binary form. Given the source code for an unsafe block, TRUST applies *software fault isolation*. Firstly, to handle unsafe heap objects in the code, TRUST identifies their allocation sites and replaces the sites with invocations to our customized allocator, which allocates memory objects in the unsafe region with a predefined address range. Next, as for unsafe stack objects, TRUST transforms the code to allocate them separately in a special stack which is positioned in the unsafe region. Finally, all memory operations in an unsafe block are instrumented with masking to deny out-of-region access. In contrast, to deal with external libraries in binary form, TRUST applies an isolation technique based on Intel *Memory Protection Key* (MPK) since we have to conservatively presume that all those library functions are unsafe and use unsafe objects only. As a result, TRUST assigns separate keys to memory pages in the safe and unsafe regions, and switches access permissions every time execution flows to and from external libraries.

We have implemented TRUST[1] by extending the Rust compiler and its runtime libraries. To evaluate the efficiency, we used fifteen widely-used crates, including core components of Rust's standard libraries. Experimental results show that TRUST slows down the test libraries by 12.65% on average. We further evaluate TRUST by comparing it with two existing techniques, namely, XRust and Sandcrust. The experimental results show that TRUST is about three times faster than XRust and more than twice faster than Sandcrust.

[1] Once published, TRUST will be open-sourced to benefit future studies.

2 Backgrounds

Ownership in Rust. Rust's ownership policy enables the compiler to obviate memory-safety bugs statically. In a Rust program, a memory object must be *owned* exclusively by one variable at a time during execution. When the program needs to copy or move a pointer to a memory object, the ownership must either be transferred to the new variable permanently or be *borrowed* temporarily. The ownership policy is stringent with zero tolerance, being imposed on data representation, abstraction, and algorithm design. For example, Rust has tree-shaped linked data structures and completely disallows the implementation of mutable data structures like doubly-linked lists. As a workaround to this strict policy, Rust incorporates unsafe Rust, a dialect that allows some relaxed performance and expressiveness rules. Unlike the ones written in safe Rust, programs written in unsafe Rust may manipulate raw pointers, invoke external library functions, or use mutable global variables. In fact, a Rust program is typically composed partly of unsafe Rust code in many forms. A code block can be wrapped with the keyword `unsafe`, or an entire function can be written in unsafe Rust when the function is annotated with the same keyword. Although an unsafe block conventionally refers to the former in Rust, we abuse the term to refer to any code written in unsafe Rust for brevity.

Smart Pointers. Smart pointers are a widely used concept in which a pointer is represented as a composite data type containing the memory address and metadata. Most commonly, the metadata is either the range of addresses that the pointer is expected to target or the pointer's capability. Many standard libraries in Rust use these smart pointers to ensure memory safety that cannot be checked statically at compile time [13]. For this reason, operations on smart pointer metadata are considered unsafe and thus can only be done in unsafe blocks.

Memory Protection Keys. Intel MPK [22], also referred to as Protection Keys for Userspace (PKU), is a per-thread hardware mechanism provided by Intel to help maintain memory page permissions in groups [37]. With MPK, each page is assigned a 4-bit value, called *pkey*, indicating the group to which the page belongs. A processor with MPK has a special register called `pkru` that determines the permission that the current process has for each page group. The permission can be investigated and updated by using special instructions, `rdpkru` and `wrpkru` [22]. The quick switch in permission using an in-process register motivated many earlier studies to use it for in-process isolation [24, 26, 44, 47] as TRUST does to quarantine external libraries. In particular, TRUST creates separate memory regions for each component, uses

```
1  static mut offset_in: i64 = 10;
2  fn main(){
3    ...
4    let array = [1,2,3,4,5];
5    let secret_code = 12345;
6    unsafe {
7        let ptr = array.as_ptr().offset(offset_in);
8        std::ptr::write(ptr, 10) }
9    let vector = vec![1,2,3,4,5];
10   unsafe {
11       let ptr = vector.as_ptr().offset(offset_in);
12       std::ptr::write(ptr, 10) }
13   ... }
```

Figure 1: Potential memory vulnerabilities caused by unsafety

```
1  fn next(&mut self) -> Option<(A::item, B::item)>{
2      if self.index < self.len {
3          ...
4      } else if A::may_have_side_effect() &&
5                self.index < self.a.size() {
6          let i = self.index;
7          self.index += 1;
8          unsafe { self.a.__iterator_get_unchecked(i); }
9          None
10     } else { ... }}
11 fn size_hint(&self) -> (usize, Option<usize>) {
12     let len = self.len - self.index;
13     (len, Some(len)) }
```

Figure 2: A buffer overflow vulnerability from unsafe Rust [7]

MPK keys and PKRU to grant and revoke access rights, and adopt the existing mechanisms needed to protect this MPK-based protection against the targeted attacks. For example, TRUST uses static analysis and carefully designed entry/exit gates as presented in ERIM [47] and Hodor [24]. Additionally, TRUST monitors and hooks system calls to further prevent the external libraries from escaping the quarantine as suggested in another work, PKU Pitfalls [20].

3 Motivation

This section gives examples of the vulnerabilities in untrusted code undermining the memory safety of safe Rust, introduces how in-process isolation mechanisms help, and discusses the limitations of existing mechanisms.

3.1 Vulnerabilities in Untrusted Code

As long as the safe Rust code and untrusted code run in the same process, Rust programs are always vulnerable to memory safety bugs [4, 7, 16] no matter how careful the compile-time analysis is. As one class of the untrusted code, unsafe blocks written in unsafe Rust are out of the scope of the analysis that is designed solely for safe Rust. Even more useless is this analysis when dealing with the other class of untrusted code, external libraries, which can be written in any language, including unsafe ones like C/C++. Such incompetence of compile-time analysis in guaranteeing memory safety of untrusted code may open doors for bugs of untrusted code to

```
1  fn overflowed_zip(arr: &[i32]) ->
2    impl Iterator<Item=(i32,&())>{
3      static UNIT_EMPTY_ARR: [(): 0] = [];
4      let mapped = arr.into_iter().map(|i| * i);
5      let mut zipped = mapped.zip(UNIT_EMPTY_ARR.iter());
6      zipped.next();
7      zipped }
8  fn main(){
9      let arr = [1,2,3];
10 let zip = overflowed_zip(&arr).zip(overflowed_zip(&arr));
11     dbg!(zip.size_hint());
12     for(index, num) in //results in stack/heap overflow
13       zip.map(|((num, _), _) | num).enumerate() {
14         println!("{}: {}", index, num);
15     } }
```

Figure 3: The vulnerability in Figure 2 being exploited [7]

```
1  pub fn printw(s: &str) -> i32{
2    unsafe {ll::printw(s.to_c_str().as_ptr()) }
3  }
```

Figure 4: Format String Vulnerability CVE-2019-15546 [4]

corrupt critical memory objects in safe blocks. Moreover,

Figure 1 shows a Rust program where a pointer is defined in a safe block and used in an unsafe block. The example allocates two stack objects at lines 4 and 9, and takes raw pointers targeting the objects at lines 7 and 11, respectively. The raw pointers are incremented at the same lines and then used for modifying the stack objects at lines 8 and 12. The problem in this example is that the program uses a global variable, `offset_in`, which could be modified externally anywhere else. This use of a global variable in offset computation renders the program vulnerable. At lines 8 and 14, an attacker manipulating `offset_in` can control which address is written to, even reaching the safe objects. Figure 2 shows another example extracted from the `zip` crate referenced in CVE-2021-28879 [7], where the `self.index` may be set to a value greater than `self.len` (line 7), resulting in an integer overflow in the `size_hint` function (line 13). Corrupting the return value of `size_hint` misleads `__iterator_get_unchecked(i)` called at line 8 to erroneously return an iterator to a memory location outside the object `a`. An attacker could exploit this to create a buffer overflow when a consumed Zip iterator is used again as in Figure 3.

Figure 4 is an example showing an external library call that may undermine the memory safety of safe objects on the execution stack. The resulting vulnerability is similar to a format string bug in C/C++. It reads as much information from the stack as there are undefined string conversion parameters (e.g., `printw("%s%s%s")`). Such a bug could pave a away for attackers' exploiting external library code to read as much information as they want from the stack and print it to the standard output.

3.2 Mitigation by In-Process Isolation

A Rust program is composed of two distinct pieces of code: untrusted code with potential exploits just discussed in the

Table 1: Comparison of In-Process Isolation Policies

	Full Automation	Protection from: Unsafe Rust Stack	Unsafe Rust Heap	External Libs Stack	External Libs Heap
XRust [31]	✗	✗	✓	✗	✗
Sandcrust [28]	✗	✗	✗	✓	✓
Fidelius Charm [15]	✗	✗	✗	✓	✓
TRUST	✓	✓	✓	✓	✓

examples above and safe blocks to be protected from such exploits. In-process isolation that isolates safe blocks and their associated data from the rest parts of the code, *i.e.*, quarantining the untrusted code, is a natural fit to solve this protection problem within a program. TRUST automatically identifies the objects that need to be protected from the untrusted code and applies in-process isolation mechanisms [27] to make sure that the untrusted code has no access to them.

XRust [31] and Fidelius Charm [15] take a similar approach as TRUST at run time. XRust enables developers to quarantine unsafe blocks using SFI. All memory access instructions marked unsafe by the programmer are instrumented with bounds checking to enforce the designed policy. Unsafe blocks can access only the unsafe objects allocated by XRust's additional memory management interfaces. They later show that additional memory management interfaces can be inserted automatically by interprocedural data-flow analysis. They also introduce an alternative to SFI, guard page-based protection, which trades security for performance. Fidelius Charm specializes in in-process isolation for quarantining untrusted external libraries from a Rust program. The kernel is extended to provide a system call interface for switching privilege levels. The protected Rust program uses this interface to switch the privilege level when entering or leaving an external code block. Upon each request, the kernel extension changes the access permission to certain memory pages containing safe objects by updating the page table attributes. Thus, developers are responsible for using the interface to augment their programs to protect sensitive data objects.

3.3 Limitations of Existing Mechanisms

Even in a combined form, these tools are inadequate to confine the attacks on untrusted code strictly, not to mention that none of the existing mechanisms quarantine both external libraries and unsafe blocks as shown in Table 1. First, most mechanisms require manual program changes. Fidelius Charm and Sandcrust do not demonstrate any automated transformation and leave it as a task for developers. XRust used existing data-flow analysis to transform the programs automatically, but the automatic transformation does not consider wrappers for heap allocators and smart pointers that Rust uses. Second, existing mechanisms require expensive context switches when entering and leaving the quarantined untrusted code context. XRust does not explicitly address the unsafety introduced by external libraries, and Fidelius Charm changes the attributes of page table entries for every context switch. This context switch is expected to exhibit long latency because the attributes of many page table entries must be updated, and the corresponding entries must be flushed. Third, Fidelius Charm, the existing in-process isolation mechanism for external libraries, does not ensure the integrity of the stack pointer. There are many existing in-process isolation mechanisms [19,39,47] in which the default context has the lower privilege, and a process temporarily enters a context with higher privilege. Unlike these, Fidelius Charm and TRUST create a context with lower privilege and give full control of the register content temporarily. This leaves the register containing the stack pointer unprotected, and an attacker may forge a *counterfeit stack* and make the stack pointer target it to affect the behavior of safe blocks. Finally, the sandboxing must also consider the stack objects because an attacker corrupting stack objects could mislead the safe Rust to violate memory safety.

Manual Analysis and Transformation. Existing mechanisms require manual program changes. Developers are supposed to identify the objects that untrusted code uses and handle them with the newly proposed memory management functions. XRust [31] is an exception in that it demonstrates how existing data-flow analysis can be used to instrument memory operations involving unsafe objects, but the application of Xrust primarily requires manual code changes, as we found in the open implementation of XRust [12]. For example, to harden an earlier example shown in Figure 1, the developer is supposed to notice that `vector` is used in an unsafe block and change the code to allocate memory from the unsafe region. Without this, XRust will not recognize the unsafety and place the `vector` in the safe region.

Inefficiency in Sandboxing External Libraries. Existing mechanisms have high context switch overhead when entering or leaving an external library. Sandcrust [28] uses a separate process running with different virtual address spaces to serve external libraries for the main process. The Rust-written code can use inter-process communication (IPC) to invoke an external library function instead of its original form, a simple function invocation. This remote procedure call obliviously increases the overhead because the arguments and return values must be passed over IPC. Moreover, the developer is responsible for translating the program manually, *i.e.*, for delivering whatever data object an external library needs or updates to and from the external library. Fidelius Charm [15] does not require IPC, but the program still needs to ask the OS kernel for a context switch. Unfortunately, this call to the kernel is still expensive because it updates the page tables to change the permission that the process has for its memory pages. The page table updates change the attributes of Rust program's pages, thereby invalidating the corresponding translation lookaside buffer (TLB) entries and increasing the TLB miss rate.

4 Threat Model and Assumptions

We make no assumptions about the untrusted code, which includes unsafe blocks and external libraries. The trusted computing base (TCB) of TRUST includes three components at run time, which are the safe blocks we assume to be trusted , the Rust runtime, and TRUST runtime. Untrusted code may have memory corruption vulnerabilities or a random series of vulnerabilities that an attacker can chain to make arbitrary memory access. Such an attacker can also compose a gadget chain to execute an arbitrary code within the context of untrusted code. We consider remote attackers aware of these vulnerabilities in untrusted code with which the victim program runs. The attackers aim to corrupt the safe objects and make safe blocks misbehave by exploiting the vulnerabilities. Attackers aware of vulnerabilities in external libraries used by a Rust program may exploit them to access otherwise privileged memory meant for safe Rust use only. Developers are assumed to know these assumptions and the benefits of using TRust, to ensure that security-critical memory objects should be used only in safe blocks. If the above assumptions hold, TRUST guarantees that external code cannot read from or write to safe objects used only by trusted code.

5 Design and Implementation

TRUST considers any memory object unsafe if it is used in untrusted code consisting of unsafe blocks of Rust, and foreign function interface(FFI) through which external library codes written in other languages such as C/C++ are facilitated in conjunction with Rust. The objects that TRUST can prove statically to be untouched by untrusted code are classified as safe. TRUST isolates unsafe objects in a separate region such that exploitation in untrusted code does not affect outside the region. Only safe blocks are allowed to access both memory regions, whereas external code can not read from or write to safe objects, and unsafe blocks can not write to safe objects. To quarantine untrusted code, TRUST analyzes and transforms a Rust program at compile-time and runs the resulting program with its runtime library.

Analysis and Transformation. TRUST first collects the Rust-specific attributes during compilation from the Rust source code to LLVM IR for the later stages. The attributes include the *unsafety* information that indicates if an LLVM IR instruction belongs to an unsafe block or not and function API information indicating whether a given function belongs to a foreign function interface(FFI). The compiled LLVM IR code is then passed to the points-to and value-flow analyses stage. This stage identifies pointers used in unsafe memory operations and performs necessary actions to isolate them. It marks instructions accessing unsafe stack objects for later relocation and reroutes heap pointer allocation calls to an unsafe allocator. The output from this stage is then passed to the LLVM compiler, which runs several passes to finalize TRUST's static operations. The final stage first relocates all unsafe stack pointers to the unsafe object stack and finally inserts *entry and exit gates* around external library calls.

Runtime. To serve the transformed program and properly quarantine the untrusted code, TRUST hooks several system services and maintains *per-thread metadata*. The heap allocator calls from untrusted code are rerouted to the additional, unsafe heap allocator that serves the requests with the chunks in the unsafe region. TRUST further hooks the memory management system calls from the external libraries to prevent them from altering page table attributes. It also augments `pthread` to initialize or destroy stacks for additional threads.

Challenges. While designing TRUST, we encountered three noteworthy challenges. Firstly, the memory safety of safe Rust is not entirely provable at compile-time. For the memory accesses that the compiler cannot reason about, such as those to heap objects, Rust uses *smart pointers* to ensure spatial and temporal safety dynamically at runtime. For this reason, manipulation of either the pointers or smart pointer metadata in unsafe blocks may result in memory-safety bugs in safe blocks [5, 6]. When identifying the objects or pointer uses that are potentially unsafe, TRUST must take metadata of them into consideration. Secondly, unlike the unsafe blocks, external libraries cannot be isolated with SFI. Such external libraries are assumed to be delivered in the form of executable binaries such as shared objects or static libraries. TRUST cannot analyze and transform them at compile-time regardless of the language that an external library is written in. Some studies have shown that SFI can be applied to binary programs, but they incur relatively high overhead [51]. For this reason, TRUST instead relies on Intel MPK mechanism to isolate the external libraries and restrict their memory access. Finally, Rust's allocation of heap memory through the `Alloc` crate and handling of heap pointers through smart pointers and container crates such as `Box, Vec, String` possess a challenge to points-to and value-flow analysis. Instead of direct invocation of the heap allocator for memory, Rust relies on these crates to invoke the allocator and create smart pointers. The `Alloc` crate exposes functions such as `__rust_alloc, __rust_realloc, __rust_alloc_zeroed`, wrapping the corresponding heap allocator interfaces. With this design, all heap pointers (safe and unsafe) appear to be aliasing only a handful sources and a single sink. TRUST handles this issue by utilizing the value-flow analysis to construct a call stack for the allocation of unsafe pointers. Then it creates a clone of the call stack, rerouted to the unsafe allocator.

5.1 Points-to Analysis

TRUST analyzes a Rust program at *Mid-level IR (MIR)* and LLVM IR level to find the allocation sites for the unsafe objects. TRUST considers an allocation site as safe and classifies an allocation site as unsafe if it finds a flow from the site to a memory access instruction in an unsafe block or to an external

library. An allocation site remains safe only if TRUST can soundly conclude that the pointer obtained from the allocation site never flows to external libraries and used for writing in unsafe blocks, as we describe in the rest of this section.

MIR-level Analysis. TRUST associates each LLVM IR instruction with the block that the instruction belongs to for later analysis while the Rust compiler translates the program from MIR to LLVM IR. Rust compiler first translates the source code to MIR, and performs the Rust-specific static analysis at the level. For example, rules related to ownership are mostly checked at that level. For this reason, each MIR instruction is tagged with the block that it belongs to, either safe or unsafe. However, this tag is not retained when the program is translated to LLVM IR because the Rust compiler does not need to determine which block is an LLVM IR instruction generated from. On the contrary, TRUST needs to distinguish the LLVM IR instructions generated from unsafe blocks for its analysis. To this end, TRUST extends the Rust compiler to attach *unsafety metadata* to LLVM-IR instructions generated.

Points-to Analysis. TRUST performs points-to analysis to classify the memory allocation sites (*i.e.*, `alloca`, calls to `malloc` or similar) into safe and unsafe. We consider `alloca` instruction as a source of pointers as well to take the stack objects into consideration. A context-sensitive value-flow analysis, SVF/SUPA [42, 43], backs our points-to analysis for improved precision. TRUST iteratively performs bottom-up value-flow and points-to analysis to obtain precise yet sound points-to relations and classify the allocation sites using the result. This bottom-up analysis is a graph traversal problem, where nodes are pointers, and edges are the instructions using them. In this sense, a program under points-to analysis can be viewed as a *Value-Flow-Graph (VFG)*, as defined by SVF/-SUP [42, 43]. An allocation site is classified as unsafe if it produces a pointer that flows to at least one instruction in an unsafe block. Otherwise, an allocation site is classified safe which means that it is within a safe block, and the pointer it produces is not manipulated or used in an unsafe block. Even if a memory object is shared between the safe and unsafe blocks, the object will be allocated from the unsafe region because the pointer's allocation site becomes unsafe. This policy makes the decision of TRUST to allocate an object from a safe region to be sound, *i.e.*, memory objects that untrusted code may access are always allocated from the unsafe region. A drawback of this approach is that a memory object that only safe code accesses could be classified as unsafe if TRUST fails to distinguish the object from a different, unsafe one due to the limited preciseness of points-to-analysis.

Modifying SVF. SVF does not handle some LLVM IR instructions, such as `InsertValueInst` and `ExtractValueInst`, on which Rust heavily relies to pack and unpack smart pointers. SVF considers any pointers entering `InsertValueInst` as entering a black hole because the instruction yields a simply packed `struct`, which is not a pointer. Similarly, any pointer yielded from `ExtractValueInst` appears to originate from a black hole. We modified SVF to handle these cases as `GetElementPtr`-related instructions followed by loads or stores. Additionally, SVF does not handle `IntToPtr` and `PtrToInt` instructions, which are used for pointer arithmetics. We address this by modifying SVF to create virtual links from `PtrToInt` instructions to an `IntToPtr` instructions using virtual pointer casts which SVF treats as pointer copies because these instructions are usually in close vicinity.

```
1 fm read_vec(idx: usize, vec: &mut Vec<i32>){
2     unsafe{ vec.set_len(idx+1); }
3     vec[idx] = 256;
4     println!("The container has been hacked: {}",vec[idx]);
5 }
```

Figure 5: Unsafe Blocks Allow Programmers to Manually Modify Smart Pointer Metadata [6]

Working budget. Although context-sensitive analysis is precise, for large programs, it becomes notoriously expensive. Therefore, SVF defines a working budget as the maximum number of contexts and edge back traversals for a given node. Additionally, instead of analyzing all pointers, we only consider pointers used in unsafe blocks as candidates. For every candidate pointer, we iteratively traverse the VFG backward until following all possible paths until either the contexts budget is exhausted or we arrive at possible allocation sites. In the event that the budget is exhausted, rather than terminating the traversal for the pointer under consideration, we relax the preciseness and fall back to flow-sensitive analysis constrained by the edges budget. Users may provide a trade-off between preciseness and time by providing the context and flow budget parameters to TRUST to use for analysis. We further discuss the impact of this design on the completeness and soundness of the analysis in §6.2.

Handling Smart Pointers. We found that, even if the points-to analysis does not report any modification of the pointer in an unsafe block, the use of the pointer could become unsafe due to the design of Rust runtime. Some pointers in Rust are represented using smart pointer types that include attributes for several purposes, including dynamic bounds checking at run time. Manipulating such attributes could lead to out-of-bound memory access within a safe block because the dynamic bounds checking using the corrupted attributes fails to find the problem, as an example (Figure 5) suggests. Our analysis overcomes this by considering an aggregate allocation site as unsafe if any field of the allocated object is potentially manipulated in an unsafe block and marks the corresponding allocation site to be unsafe.

Allowlisting Crates. While analysing unsafe pointers, TRUST considers some unsafe blocks from certain, allowlisted crates (e.g., `libstd, liballoc, libcore`) as safe. In other words, the use of a pointer in an unsafe block from any of these crates alone does not render it a candidate

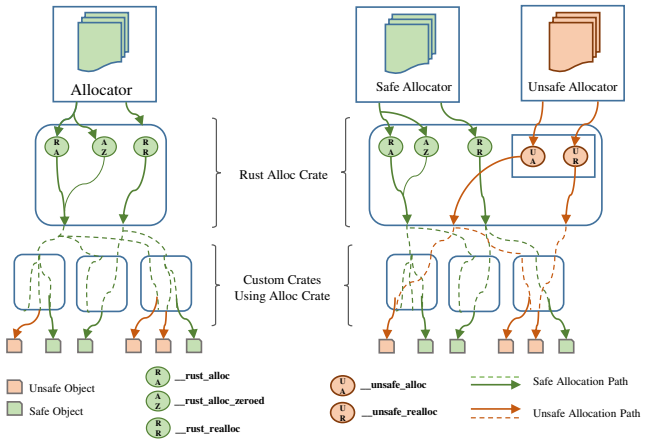

Figure 6: Automatic Function Cloning and Heap Allocation Context Stack Replacement.

for points-to analysis or relocation. We find this an essential yet reasonable design choice for two reasons. First, the core of Rust is mostly synced to the kernel, requiring system calls and low-level operations that must be wrapped in unsafe blocks. If we consider all these untrusted, a Rust program will have virtually no safe objects, and this makes TRUST unfruitful. Second, a recent study has shown that such critical crates can be formally verified [25]. The unsafe blocks in core runtime could not be written using safe Rust, but statically verifying their correctness and thus trustworthiness as safe is still possible, using available mechanisms such as ‘ [25]. This backs our decision to include them in the allowlist. Note that TRUST is still needed despite this static verification because not every untrusted code can benefit from the targeted static verification.

5.2 Function Cloning to Improve Precision

As explained earlier, among the challenges faced by TRUST is the small number of heap allocation sites at the LLVM IR level. All heap pointers appear to originate from a handful of allocation sites in the wrapper function in LLVM IR, and this leads existing value-flow analysis §5.1 to conclude that all pointers share the few sources and sinks discovered. TRUST addresses this problem by automatically cloning the functions that handle both safe objects and unsafe ones depending on the context, to improve the precision of points-to analysis. This is done by the following three steps.

Step 1. Assigning Call Site-IDs to VGF Nodes. Based on SUPA [43], TRUST's VFG creates two sets of virtual nodes, namely, *in-nodes* and *out-nodes* for every call site. In-nodes connect actual pointer arguments taken by the call to the callee's formal parameters, while out-nodes include copies of any pointers either returned through an actual return statement or stored in any of the taken arguments. Both in-nodes and out-nodes are assigned the same call site-ID (cs-ID), which in turn is associated with the caller and callee functions. If the

```
;;original store in unsafe block
{
    store i32 10, i32* %ptr, align 4, !UNSAFE_BLOCK
}

;;transformed bitcasted store
{
    %temp = ptrtoint i32* %ptr to i32,  !UNSAFE-BLOCK
    %masked_ptr = and i32 %temp, %UNSAFE_MASK,  !UNSAFE-BLOCK
    %bitcasted = bitcast i32 %masked_ptr to i32*,  !UNSAFE-BLOCK
    store i32 10, i32* %bitcasted, align 4, !UNSAFE_BLOCK
}
```

Figure 7: Instrumenting Store Operations in Unsafe Blocks at LLVM IR Level

callee takes no pointer arguments, all call sites it is associated with will have empty in-nodes. The same is true for returned pointers and out-nodes.

Step 2. Finding Clone Candidates. TRUST finds the function to clone by backward-traversing the VFG obtained during points-to analysis, until it arrives at a call site to a heap allocation (*i.e.*, `malloc`). It pushes the cs-IDs of in-nodes and out-nodes that it encounters during the traversal to a stack called *call site stack*. It then solves a *balanced-parentheses problem* [43] on the call site stack, where pairs of in- and out-nodes with matching cs-IDs are removed as they don't contribute towards the actual allocation path, leaving cs-IDs of unbalanced in-nodes and out-nodes. Finally, it also removes the in-node cs-IDs, as these represent caller functions, yet TRUST's goal is to replace the callee with a cloned function that leads to the unsafe allocator. The remaining set of consecutive out-node cs-IDs trails to the heap allocation site. All callee functions associated with these remaining cs-IDs are cloned, with the cloned versions having `__unsafe` prefixed names and an additional argument.

Step 3. Transforming the Call Sites. After cloning the functions, TRUST reconstructs the allocation path by recursively transforming appropriate call sites. TRUST replaces a call site if its cs-ID remains in the call site stack, with a call to the clone of callee function. The additional argument is the bit vector where each bit indicates whether the corresponding call site must route to the unsafe allocator. For indirect calls whose callee is a member of a virtual table, instead of the additional argument, a bit flag, propagated through the `unsafe_flag` entry in the thread-specific data structure, is used. Figure 6 shows how this cloning helps improve the precision of the analysis. Initially, all pointers—safe and unsafe, allocated from the safe allocator (paths indicated by green dotted lines). After the cloning by TRUST, unsafe pointers are distinctively allocated from the unsafe allocator (paths indicated by the brown dotted lines).

5.3 Instrumenting Memory Accesses for SFI

TRUST instruments the store instructions in unsafe blocks to prevent them from directly corrupting the safe objects. In particular, TRUST inserts instructions that mask the address

```
define void @"write_to_pointer_offset"(
    {i8*, i64} %sptr, i8 %idx, i8 %data)
{
    %temp = getelementptr inbounds { i8*, i64 },
      { i8*, i64 }* %sptr, i32 0, i32 0
    %ptr = load i8*, i8** %temp, align 8
    %offset = getelementptr inbounds i8, i8* %ptr,
      i8 %index
    store i8 %data, i8* %offset, align 4;;maybe hazardous!
    ret void
}

;;changing metadata of a smart pointer
{
    call void @change_smart_ptr_metadata(
        %smart_ptr), !UNSAFE_BLOCK
}

;;CASE 1: Use in same function as metadata change
{
    %temp = getelementptr inbounds { i8*, i64 },
      { i8*, i64 }* %sptr, i32 0, i32 0
    %ptr = load i8*, i8** %temp, align 8
    %offset = getelementptr inbounds i8, i8* %ptr,
      i8 %index
    store i8 10, i8* %offset, align 4;;hazardous!
}

;;CASE 2: Use in callee function
{
    call void @write_to_pointer_offset(
        %smart_ptr, i8 %index, i8 10)
}
```

(a) Original Uninstrumented Code Snippet

```
;;changes to handle CASE 2
define void @"write_to_pointer_offset"(
    {i8*, i64} %sptr, i8 %idx, i8 %data) {
    %temp = getelementptr inbounds { i8*, i64 },
      { i8*, i64 }* %sptr, i32 0, i32 0
    %ptr = load i8*, i8** %temp, align 8
    %test = call i1 @is_in_unsafe_region(i8* %ptr)
    %check = icmp eq, %test, true
    br i1 %check, label %unsafe_handle, label %safe_handle
unsafe_handle:
    %offset = getelementptr inbounds i8, i8* %ptr, i8 %index
    %bitcasted1 = bitcast i32* %offset to i32
    %masked_ptr = and i32 %bitcasted1, %UNSAFE_MASK
    %bitcasted2 = bitcast i32 %masked_ptr to i32*
    store i8 %data, i8* %bitcasted2, align 4;;safe
    br label %end
safe_handle:
    %offset = getelementptr inbounds i8, i8* %ptr, i8 %index
    store i8 %data, i8* %offset, align 4
    br label %end
end:
    ret void
}
;;changes to handle CASE 1
;;CASE 1: transformed
{
    %temp = getelementptr inbounds { i8*, i64 },
      { i8*, i64 }* %sptr, i32 0, i32 0
    %ptr = load i8*, i8** %temp, align 8
    %offset = getelementptr inbounds i8, i8* %ptr,
      i8 %index
    %bitcasted1 = bitcast i32* %offset to i32
    %masked_ptr = and i32 %bitcasted1, %UNSAFE_MASK
    %bitcasted2 = bitcast i32 %masked_ptr to i32*
    store i8 10, i8* %bitcasted2, align 4;;safe
}
```

(b) TRUST Instrumented Version

Figure 8: Instrumenting Pointer Offsets From Vulnerable Smart Pointer Metadata

before these store instructions to enforce that the instructions are writing only to the unsafe region as shown in Figure 7. The static bound that the inserted instructions use is determined when the program starts. In addition, TRUST instruments some store instructions in the safe block as well to address the *confused deputy* problem. As discussed earlier, some smart pointers are allocated on the unsafe region because they are legally modified in unsafe blocks. This exposes their metadata to arbitrary corruption by an exploit, and this corruption may confuse the safe block that uses the metadata for bounds checking. TRUST automatically identifies such vulnerable pointer flow within the safe block and inserts the bounds check to ensure that the pointer targets within the unsafe region. To find the vulnerable pointer flows, TRUST uses the value-flow graphs. TRUST first finds the paths in the value-flow graph from an unsafe pointer to a store instruction in the safe blocks. Along such a path, TRUST inserts a bounds check that ensures that the unsafe pointer targets the unsafe region as shown in Figure 8. Lastly, the commonly used memory modification functions such as `memset`, `memcpy`, and `memmove` are specially handled instead of being instrumented within. At the call sites, TRUST inserts a bounds check to ensure the whole range of memory to be written is contained in the unsafe region if the destination pointer is unsafe.

5.4 Unsafe Object Stack

TRUST transforms the program to have three stacks. The first is the default stack that contains safe stack objects only on a program protected with TRUST. The other two are the stacks to accommodate unsafe stack objects. We call one of them for the external libraries as the *unsafe execution stack* and further describe in §5.5. Inspired by SafeStack [18], we transform the program to store unsafe objects on another stack called *unsafe object stack*. SafeStack mitigates stack-based buffer overflow attacks by moving potentially vulnerable buffers to a separate region, where their exploitation cannot recur and cannot result in more dangerous attacks such as control flow hijacking. We store the pointers to these stacks in a thread-specific metadata object whose address, in turn, is maintained in a designated register, `r15`. The compiler is augmented to set the register aside and not to use it for register allocation, and the TRUST runtime initializes `r15` with the addresses of the unsafe stacks. Accordingly, all unsafe memory access instructions to a stack object are transformed to use `r15` instead of `rsp` with the correct offset. Alternatively, TRUST can use `%gs` or `%fs` instead of `r15` to avoid performance degradation due to the potential register spills. Intel's `FSGSBASE` instructions enable programs to directly access the `fs/gs` segment registers, and TRUST can use one of these instead of `r15` if the one is not used for other purposes.

5.5 Instrumenting External Library Calls

TRUST inserts the entry gate and exit gate before and after an external library invocation to quarantine external libraries using Intel MPK. The external function, the callee, is originally responsible for saving the stack pointer on its stack and restoring it, in which the attacker can create a fake stack from scratch and set the stack pointer when returning to the Rust code. These gates update the `pkru` register to temporarily restrict memory access privileges while the external library runs, switch the stack pointer, so the external library uses the unsafe stack, and save the stack pointer in a protected memory for retrieval after returning from the external library. They also write the appropriate data entries of the thread-specific data structure for TRUST runtime (see §5.6).

Entry Gate. In the entry gate, the program saves the safe stack pointer (`rsp`) in the thread-specific data structure residing in the special region (always readable). It updates the domain entry in the data structure to 1 to indicate that the program runs in the context of external libraries, and then the `pkru` register to revoke read-write permission to the safe memory region (with pkey 0), and write permission to the always-readable special region (with pkey 2). Finally, `rsp` is set to point the unsafe execution stack for external libraries.

Exit Gate. On return from the external library, the program makes a simple call that matches the contents of the `r15` with those saved earlier by the entry gate. In case of a mismatch, TRUST considers it a violation and resets the `r15` register to the correct value. This check is important because attackers may misdirect TRUST to execute the safe block on an unsafe stack, as explained above. Next, it updates the access permission of the safe region and special page (on which the `r15` register contents were saved during entry) to read-writable. At this point, write permissions to the safe region have been restored, so TRUST updates the domain entry value of the thread-specific data to 0. Finally, it loads the saved safe stack pointer from an offset to the `r15`, and updates the `RSP`.

Comparison to SFI. A potential alternative to our design choice, the use of MPK, is SFI. Prior studies [21, 55] have reported that it is possible to apply SFI even to binary programs, and it is theoretically possible to use SFI for our purpose as well. The rationale behind our choice of using MPK is the frequency of domain changes while running external libraries and the number of memory access instructions. The performance overhead of SFI is primarily related to the total number of memory instructions because each memory instruction must be instrumented with bound checks. In contrast, the overhead of MPK is related to the number of transitions. When quarantining the external libraries, we expect to have a relatively small number of transitions, while each library call may have many memory instructions.

5.6 TRUST Runtime

TRUST runtime is composed of per-thread metadata store for TRUST in the safe region, an additional heap allocator that handles unsafe object allocations, hooks to memory management system calls to prohibit the external libraries from altering page attributes, and hooks to `pthread` that initializes the TRUST runtime for new threads. After the unsafe region is established, these components work as follows. First, the unsafe heap allocator and system call hooks ensure that any objects or pages that untrusted code allocates lie in the unsafe region. Failing to do this does not undermine security because the untrusted code is prohibited from accessing the safe region separately, resulting in false positives. Second, the `pthread` hooks prepare the additional stacks that TRUST needs for new threads and destroy them when the thread terminates. Third, per-thread metadata is used by the instrumented code and the runtime to store TRUST-specific data that must be protected from the untrusted code. Fourth, the system call hooks prevent untrusted code from altering the page attributes, which the MPK-based isolation of TRUST relies on.

Establishing the Unsafe Region. TRUST establishes the unsafe region when the program starts by obtaining a large enough number of virtual pages from the OS, and the total size of such pages is 4GB in our implementation. The pages are mapped using the flag `MAP_FIXED` and configured to have the pkey for the unsafe region. As we describe later, any demand on memory chunks or pages on the unsafe region will be served using these mappings. The approach that maps the unsafe region at the startup automatically ensures that the remainder, the safe block, will always obtain chunks or pages from the safe region because the OS does not map already mapped pages unless requested explicitly. It is worth noting that the size of the reserved address range can be increased if the program is expected to use more than 4GB for unsafe objects, and doing so will not incur performance or memory overhead. Requesting the OS kernel to reserve more memory will only set more virtual pages aside, and only the virtual pages that are actually allocated will be mapped to the physical pages. It would be even better if the OS kernel supports a different means that TRUST can use to reserve a particular virtual address range without creating a fixed mapping, but the Linux kernel that our prototype runs on does not have such a functionality, to the best of our knowledge.

TRUST Metadata. TRUST maintains one metadata store for each thread to keep TRUST-specific data and represent the privilege level (e.g., safe, unsafe, or external) for the other components of TRUST runtime. The metadata has a one-bit flag representing its privilege level, called `external`. The `external` bit is to be set when the program enters an external library and to be unset on exit. The metadata also includes the field for safe stack pointer, which TRUST uses to verify the stack pointer integrity on its exit gate as described in §5.5.

The Unsafe Heap Allocator. TRUST uses an additional,

modified heap allocator to manage a heap on the unsafe region, using `mimalloc` 1.7.0 [30]. Heap allocation requests from untrusted code are redirected to here by the function cloning (§5.2), and this allocator serves the requests with the chunks from the unsafe region. The modified allocator manages its heap on the unsafe region by obtaining the new pages for the heap from the established unsafe region. The requests for large chunks, which are typically served directly with `mmap`, are also served using the unsafe region pages. The program uses this unsafe heap in two ways. The unsafe blocks that TRUST compiles from the source code are transformed to invoke the unsafe heap allocator explicitly, whenever needed. The external libraries, on the contrary, cannot be transformed because they are delivered as executable binaries. Therefore, TRUST also inserts a hook to the safe heap allocator to selectively route a heap allocator call to the unsafe allocator. TRUST uses the `external` field in its metadata, which we described earlier in this section, to determine which heap allocator to call. The field is maintained by the entry and exit gates as described in §5.5, and this hook uses this bit to determine the current context. Note that while this bit is protected from exploits against untrusted code by being placed within the safe region, malicious corruption of the bit does not undermine the security guarantee of TRUST. The expected outcome of such corruption is to allocate a safe object to the untrusted code, resulting in a false positive. For heap memory reallocation and freeing, the runtime library checks whether the pointer in question lies in the safe or unsafe region and reroutes the calls accordingly.

Hooking System Calls. The runtime hooks memory management system calls (e.g., `mmap, mprotect, or mremap`) to prohibit the external library from manipulating page attributes arbitrarily. In particular, TRUST ensures that the start and end addresses of the memory that is mapped by the invocation lie within the aforementioned virtual address range corresponding to the unsafe region. The runtime also redirects calls to `mmap, mremap` from the safe block attempting to map memory from the specified unsafe region address space. Similar requests from external code are redirected to the unsafe allocator as `malloc` or `realloc` for `mmap` and `mremap`, respectively. The `mprotect` hook rejects any attempts from the external library to prohibit it from altering the page attributes, including the pkey. In addition, the external library is statically checked for the presence of instructions updating `pkru` using the algorithm proposed in ERIM [47].

Hooking `pthread`. The runtime library hooks the thread creation libraries to allocate the unsafe stacks and the thread-specific data structure. For new threads created by safe blocks, the hooked `pthread` function allocates the thread-specific data in the appropriate region. It then allocates the unsafe stacks in the unsafe region and writes their pointers in the respective entries of the thread-specific data structure. Finally, it updates the `r15` register with the pointer to the thread-specific data structure and lets the execution proceed. If an external library creates a thread, the runtime library ensures that the default execution stack for that particular thread is allocated in the unsafe region. The lifetime of a thread created by an external library is considered unsafe. Thus TRUST allocates its data structure on a page in the unsafe region and revokes write access permissions on it after writing the domain entry of the data structure. While executing external code, the runtime library reads the pointer to the thread-specific data by calling `pthread_get_specific` instead of reading the `r15` register. TRUST does this because it has no control over the `r15` register in the external untrusted domain.

Hooking More System Calls The TRUST runtime must hook more system calls to defend against attacks that a recent study [20] enumerated. For example, `sigreturn` is a system call that can be used to bypass conventional PKU-based in-process isolation techniques by corrupting the content of `pkru` register on the stack. TRUST can prevent this by storing `pkru` before the system call so that any attempt to tamper with the register is invalidated. Similarly, TRUST can naturally adopt the hooks that the earlier study found essential to further harden safe Rust from more attacks exploiting vulnerable external libraries. Besides the above, the runtime must hook more system calls to thwart the isolation bypasses, as presented in the earlier study [20]. Jenny [38] has already shown how we can comprehensively filter the system calls using `seccomp` and `ptrace`, so we can combine TRUST runtime and Jenny for completeness. The additional overhead coming from the use of Jenny is expected to be less than 5%, according to Jenny's reported overhead.

6 Evaluation

This section shows the performance impact of TRUST, presents the result of our security analysis, and discuss the precision and soundness of the static analysis.

6.1 Performance

Experimental Setup. We implement the analysis and transformation for TRUST on Rust 1.49 and LLVM 11.0.1. We run the benchmarks on a system with an Intel i9-10900K CPU, 128GB main memory, and Ubuntu 18.04.6 LTS as the operating system. We use eleven Rust crates that XRust used for comparative study with it, in execution time, memory usage, and effectiveness. For a fair comparison, we use the SFI version of XRust instead of the guard-page version because the guard-page version assumes contiguous access and is vulnerable to the *jump* accesses. We also compare the execution time of TRUST against Sandcrust using Snappy [14] as the benchmark because Sandcrust is evaluated with it. Lastly, we use two large Rust crates to demonstrate that TRUST can analyze and protect real-world software written in Rust. Note that we could not apply XRust to these because it is required to identify the unsafe objects manually to use XRust. For a similar reason, we could not compare TRUST with Fidelius

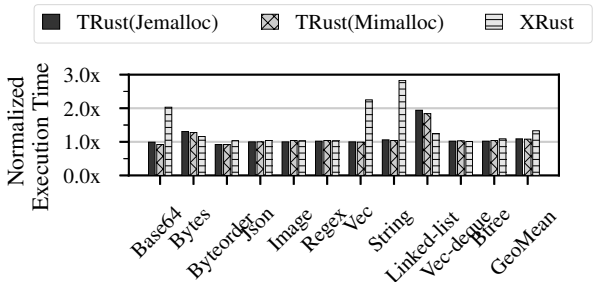

Figure 9: Normalized execution time of TRUST and XRust tested with the 11 widely used crates.

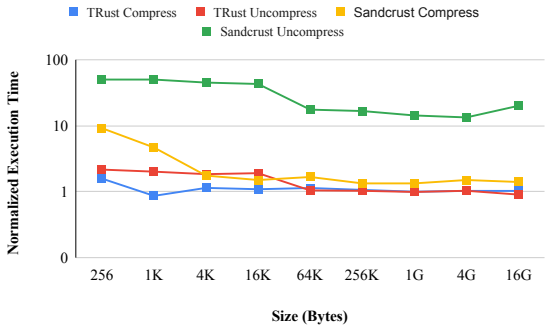

Figure 10: Normalized execution time of Snappy with TRUST and Sandcrust. The y-axis is log scale.

Charm [15] that requires developer annotation. For comparison study, we primarily use figures and present the absolute numbers in Appendix A.

Execution Time. As Figure 9 shows, the overhead of TRUST is about 7.55% on average (geometric mean) while XRust induces 26.39%, albeit the fact that TRUST also protects safe objects on the stack and quarantines the external library. TRUST exhibits lower performance overhead despite its stronger protection thanks to its optimized implementation of SFI where it uses the address masking instead of tests and conditional branches used by XRust. Figure 10 shows the result of running snappy with TRUST and along with the overhead of Sandcrust. The overhead of TRUST is 8.79% for compress and 35.12% for uncompress on average, and this is much lower than those of Sandcrust, 107% for compress and 2596% for uncompress, even though Sandcrust quarantines the external libraries only.

Memory Overhead. We measure the maximum resident set size during the execution of a benchmark to evaluate the memory overhead. We use the executions without protection using the default allocator (`glibc`) as the baseline, and present the relative memory usage of the unprotected executions with `mimalloc`, TRust with `jemalloc`, TRust with `mimalloc`, and XRust. TRust with `jemalloc` and `mimalloc` uses `glibc` as its safe allocator and the latter as the unsafe allocator. As Figure 11 shows, TRUST uses 35% more memory on average when it uses `jemalloc` as its unsafe allocator,

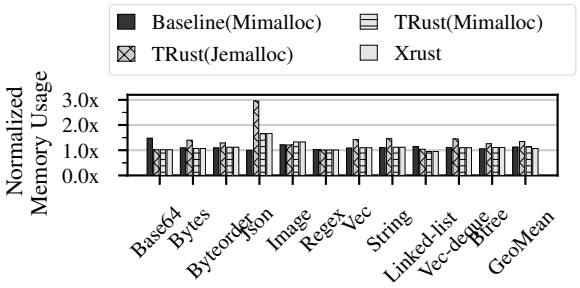

Figure 11: Normalized memory usage of TRUST and XRust tested with the 11 widely used crates.

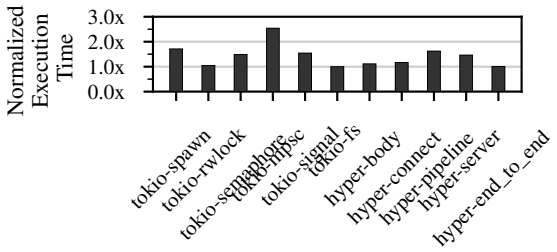

Figure 12: Performance Overhead of TRUST on Tokio/Hyper.

Table 2: Result of Using TRUST to mitigate the known vulnerabilities.

CVE ID	Origin	Affected Memory	TRUST	XRust
2021-29939	Unsafe Rust	Stack	✓	✗
2019-15546	External Lib.	Stack	✓	✗
2021-28879	Unsafe Rust	Stack, Heap	✓	Not Tested
2021-28028	Unsafe Rust	Heap	✓	Not Tested
2021-45707	Unsafe Rust	Heap	✓	Not Tested
2018-1000657	Unsafe Rust	Heap	✓	✓
2018-1000810	Unsafe Rust	Heap	✓	✓

and the overhead goes down to 13% with `mimalloc`, while XRust induces 7% memory overhead.

Large Programs. To show that TRUST can harden large programs, we use it to quarantine the untrusted code in Tokio [1] and Hyper [10]. Tokio is an event-driven, non-blocking I/O platform for writing asynchronous Rust network applications, and Hyper is an HTTP library used as a building block for many applications. Figure 12 shows TRUST's runtime performance overhead on Tokio and Hyper benchmarks. We notice that TRUST instruments a high number of instructions in these benchmarks due to the large code base, but most notably, as these benchmarks use heavy parallelism (10-1000x threads), TRUST spends a significant time creating and destroying unsafe stacks during each thread spawn. In fact, because of this, benchmarks such as `tokio-mpsc` and `tokio-spawn` required reducing the default unsafe stack size to run to completion. We also find that Tokio and Hyper depend on many external libraries, requiring frequent MPK access and execution stack switching during transition to and from FFI calls.

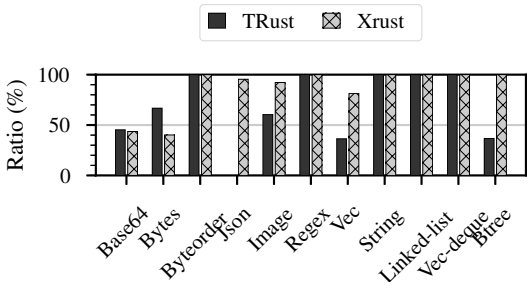

Figure 13: Ratio of the Number of Safe Heap Allocations.

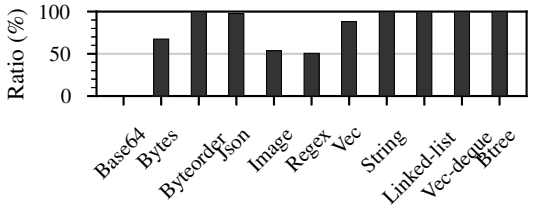

Figure 14: Ratio of the Number of Safe Stack Allocations.

```
#[inline]
fn extend_from_slice(dst: &mut Vec<u8>, src: &[u8]) {
    let dst_len = dst.len(); let src_len = src.len();
    dst.reserve(src_len);
    unsafe {
        // We would have failed if `reserve` overflowed
        dst.set_len(dst_len + src_len);XX
        ptr::copy_nonoverlapping(XX
            src.as_ptr(),
            dst.as_mut_ptr().offset(dst_len as isize),
            src_len);XX
    } }
fn write(&mut self, slice:&[u8]) -> io::Result<()> {
    extend_from_slice(&mut self.code, slice); Ok(())
}
```

Figure 15: Unsafe Code in the Json Library. XRust fails to observe the effects of this Code on the argument pointers.

6.2 Effectiveness

We evaluate the effectiveness of TRUST using synthesized exploits that are inspired by real-world vulnerabilities. We also analyze how TRUST will mitigate the other known vulnerabilities. Table 2 summarizes the result of our evaluation using real-world vulnerabilities.

Synthesized Vulnerabilities in Unsafe Blocks. We reproduced two real-world vulnerabilities, CVE-2021-29939 and CVE-2021-28879, in a small program to evaluate the effectiveness of TRUST. These two vulnerabilities are found from StackVec and Zip, respectively, and share the same root cause. Both StackVec and Zip use the `Iterator` which in turn requires the implementation of `size_hint` as in Figure 2. As already seen from the Zip crate, an integer overflow in `size_hint` returns an unsound value, which is later used to return an iterator without bounds checking, when used by the `get_unchecked` function from the `Iterator` trait. `StackVec::extend` [8] suffers from the same problem. Looping with an iterator obtained this way allows writing beyond the stack capacity as shown in [11]. Similarly, as shown in Figure 2 and Figure 3, reusing the consumed `zip` iterators introduces the stack or heap buffer overflow vulnerabilities which may affect safe Rust. In both of these vulnerabilities, the `Iterator` is accessed in an unsafe block where `get_unchecked` is executed. We reproduced these bugs in small programs and applied TRUST to them. For Figure 3, depending on where `arr` resides, it is possible to attack either the stack or heap. We make slight modifications to the code shown in Figure 3 and attempt to overwrite the stack, and then repeat the attack with `arr` of a heap-based container. The attack succeeds when the program runs without TRUST. However, TRUST stops the attack because `arr` is allocated in the unsafe region, but any attempted overflows into the safe region are prohibited by TRUST and cause the program to halt. We perform a similar attack on `StackVec` using older versions with no bug fix, and again TRUST manages to protect the safe stack. Finally, we applied TRUST to more programs with CVEs [5, 6, 9] and those [2, 3] examined by XRust. Our evaluation finds that TRUST manages to detect vulnerabilities introduced and thwart any attempts to leverage them to contaminate safe Rust.

Analysis on Vulnerable External Libraries. TRUST can also mitigate CVE-2019-15546. This vulnerability is found in `Window::printw` and `Window::mvprintw`. These functions pass a raw pointer to external functions without any sanitization, exposing users to format string bugs. TRUST identifies the objects whose pointer could be exposed and allocates them in the unsafe region, and the functions are considered untrusted for being external libraries. As a result, an attacker exploiting this vulnerability cannot alter any safe object because the operating system will trap such an attempt due to the protection key violation.

6.3 Discussion

Precision. To evaluate the precision of TRUST's analysis, we count the number of unsafe objects that TRUST finds and compare the results with the number of unsafe objects that XRust finds with the help of the developer. Figure 13 and Figure 14 shows the ratio of unsafe heap and stack object allocations when we use TRUST and XRust. Note that we do not count the number of unsafe stack object allocations for XRust because it does not protect stack objects. Out of 11 benchmarks, TRUST leaves the non-trivial amount of safe objects for 9 benchmarks despite its conservative policy that classifies any objects that the untrusted code might use as unsafe. Specifically, TRUST finds fewer unsafe objects than XRust on one (Bytes) and more on five (Json, Image, Regex, Vec, and Btree). In the case of Bytes, we found that the developer annotation for XRust on Bytes makes the unsafe allocator as the global allocator for this specific benchmark. This also demonstrates the potential imperfectness of human annotation in that any

Table 3: Number of memory access by safe Rust

Benchmark	Read From Unsafe	Ratio	Write To Unsafe	Ratio
base64	8.9G	22%	2.3G	59%
bytes	0.5G	43%	0.4G	32%
byteorder	0	00%	0	00%
json	3.8G	74%	1.4G	52%
image	5.7G	41%	1.3G	22%
regex	1.3G	4%	29.0M	81%
vec	5.8G	68%	0.7G	37%
string	0	0%	0	0%
linked-list	0	0%	0	0%
vec-deque	0	0%	0	0%
btree	81.0M	8%	0.4M	1%

Table 4: Number of memory access by untrusted code

Benchmark	Total Reads	From Safe	Total Writes	To Safe
base64	38.2G	37.0G(96%)	38.5G	0(0%)
bytes	0.2G	0(00%)	0.1G	0(0%)
byteorder	0.0G	0(00%)	0.0G	0(0%)
json	1.2G	0.7G(66%)	0.5G	0(0%)
image	0.3G	0(00%)	1.3M	0(0%)
regex	0.0G	0(00%)	12.0G	0(0%)
vec	0.0G	0(00%)	48.0G	0(0%)
string	0.0G	0(00%)	0.0G	0(0%)
linked-list	0.0G	0(00%)	0.0G	0(0%)
vec-deque	0.0G	0(00%)	0.0G	0(0%)
btree	0.7K	0(00%)	1.9K	0(0%)

Table 5: OOBudget queries from points-to analysis while using TRUST for large programs.

Benchmark	**Total Queries**	**Large Budget**			**Small Budget**	
		Compile Time (s)	OOB	False Positive	Compile Time (s)	OOB
hyper-body	27	537	8	0/0	4	9
hyper-connect	187	20	1	0/0	2	1
hyper-end_to_end	8599	>1h	-	0/19	85	730
hyper-pipeline	4399	20	0	0/1	18	124
hyper-server	5432	274	2	0/3.2M	35	155
tokio-rwlock	463	82	3	0/11.1M	5	109
tokio-semaphore	303	293	10	0/12.0M	10	73
tokio-mpsc	1387	446	151	1.9M/54.1M	11	212
tokio-signal	467	113	35	0/6.7M	3	68
tokio-fs	547	2	0	0/8	2	92

unsafe objects that XRust misses could be accessed by the untrusted code, enabling it to access the safe heap. We manually investigated the objects that TRust classifies as unsafe while XRust classifies as safe. Figure 15 shows a snippet of code from the `Json` benchmark. The `write` function, used by all benches in Json, in turn calls `extend_from_slice`, which modifies `self.code`'s pointer metadata on line 7 in an unsafe block, making `self.code` an unsafe object. However, XRust misses this and classifies `self.code` as safe. Moreover, although XRust considers read operations as unsafe, they also fail to classify `slice` arguments to `write` as unsafe as these end up being used in the unsafe block at lines 8-11. This also explains the difference in the number of unsafe objects that TRUST and XRust finds.

Soundness. We evaluate the soundness of TRUST when applied to the 11 crates and two large programs by investigating if its points-to analysis has *Out Of Budget (OOBudget)* queries. As Table 4 suggests, TRUST does not have any OOBudget queries when it is applied to the 11 benchmarks that we use for the comparison with XRust, indicating that the analysis of TRUST against these benchmarks is sound. However, TRUST experiences OOBudget queries when classifying allocation sites in large programs, Hyper [10] and Tokio [1], as Table 5 shows. The number of OOBudget queries reduces if we increase the budget parameter with the cost of increased compile time, but TRUST still have some OOBudget queries in a few benchmarks. As mentioned earlier, however, this does not render the protected program insecure in that all allocation sites are classified as safe by default. The OOBudget queries may lead to false positives because it leads TRUST to allocate a safe object when it must allocate an unsafe one. To investigate if the OOBudget queries lead to false positives at run time, we counted the number of writes to safe objects from unsafe blocks. As Table 5 shows, we find that only one out of 10 benchmarks derived from Hyper and Tokio exhibits false positives. Note that this does not guarantee that the programs will not face false positives when they run in practice. A potential remedy to this limitation is to perform memory safety checks to the OOBudget-related memory accesses, which are only a few.

TCB Size. We compare the size of TRUST's TCB with those of the others. As mentioned in §4, the TCB of TRUST at run time is composed of the core Rust runtime, the safe block, and TRUST's runtime that manages its metadata intervenes in system calls and mediates external library calls. Like TRUST, XRust also trusts the Rust runtime and the safe block, as well as its runtime library. For this reason, we quantitatively compare the TCB size of TRUST and XRust using the source lines of code (SLOC) of their runtime libraries. Surprisingly, XRust runtime is composed of 8 lines of additional code that is invoked at runtime only for bounds checking. Unlike this, TRust runtime is composed of 500 lines of code, most of which are for hooking system calls and handling thread setup and destruction to quarantine the external libraries. The allocator that XRust is using for both safe and unsafe allocation consists of 6.9k lines of code. On the other hand, two interchangeable allocators of TRUST, which are `mimalloc` and `jemalloc`, are comprised of 7k and 24.5k lines of code, respectively.

Impact of Data Flow from Unsafe to Safe. TRUST leaves data-flows from unsafe objects to safe objects because safe blocks can read from unsafe objects and write to safe objects. This may leave an attacker exploiting vulnerabilities in unsafe blocks to find and exploit some logic bugs that can be exploited only by corrupting some unsafe objects, potentially enabling to use of safe blocks as a confused deputy. For instance, a function in a safe block may use user input in an unsafe region to update a security-critical configuration that lives in a safe region. The developer may trust the user input that has passed a series of sanity checks, but an attacker exploiting the unsafe block's vulnerability corrupts the input after the sanity check to corrupt the security-critical configuration in the safe region using the function as a confused

deputy.

7 Related Work

Our work is related to the prior work on mitigating memory safety vulnerabilities, the in-process isolation mechanisms, and the mechanisms that aim to quarantine some untrusted code from safe Rust.

Mitigating Memory Safety Bugs. Researchers have strived to fight against memory safety bugs. Programs written in C/C++ are prone to such bugs due to the design of the languages and the complexity of modern software. A large body of existing mechanisms aims to mitigate the exploits of such bugs [17, 23, 29, 34, 36, 40, 54]. Compared with these, TRUST makes a unique contribution by combining and tailoring in-process isolation mechanisms to protect Rust's safe blocks.

In-process Isolation. In-process isolation is a commonly used building block to harden a program against software attacks. Among these, the SFI-based and MPK-based are most closely related to TRUST because it combines those to implement in-process isolation for a Rust program. Wahbe et al. [49] invented the design of logically separated fault domains within a single address space and instrumentation with bounds checks to prevent untrusted components' access to other components' memory. However, bound checks impose overhead on the execution of all untrusted memory accesses, even with more efficient masking-based SFI [53]. Koning et al. [27] compared the characteristics of in-process isolation techniques, including hardware based approaches, and reported that MPK is suitable for isolating or quarantining a large piece of the program due to its low permission switch latency. The advantage of MPK in permission change latency comes from the lack of supervisor calls, with the risk of leaving the permission change instruction unprotected. ERIM [47] addresses this problem by binary scanning and carefully designed call gates. It ensures that the untrusted code does not have the permission changing instruction (`wrpkru`) by binary scanning and that the instructions in trusted code are not misused by the call gates. Hodor [24] is another work on PKU-based sandboxing. While ERIM uses static binary rewriting to negate unsafe `wrpkru` instructions, Hodor relies on hardware watchpoints that the modified kernel provides. Despite these two studies' advances, a recent study [26] reported that they can still be bypassed by an attacker using the OS kernel as a confused deputy. Cerberus [48] attempts to complement ERIM and Hodor against attack vectors proposed by this study [20] except for the attack abusing signals. Jenny [38], a recent security implication for PKU-based sandbox, supports secure call gates and system call filtering along with secure signal handling to prevent the abuse of signals.

Compared with these, the contribution of TRUST is in demonstrating its efficient implementation of SFI and MPK, using analysis tailored for unsafe Rust and the external libraries, and identification and defense against the counterfeit stack attack.

Memory Safety of Rust Programs. Crust [46] translates Rust programs into C to verify the memory safety of unsafe code using bounded model checking. Sandcrust [28] is similar to TRUST in that both aim to protect Rust code and data from potentially harmful C libraries. With the help of the Rust macro system and existing sandboxing techniques, function calls annotated by the programmer are translated into RPCs. Like TRUST, XRust [31] and Fidelius Charm [15] aim to isolate the safe part of Rust from untrusted code. The latter uses `mprotect` system calls to isolate programmer-specified data from unsafe foreign libraries. Unlike Fidelius Charm, which requires invasive modification of source code, the former utilizes a separate heap allocator to isolate the safe code of Rust from unsafe heap objects without user annotation. CLA [33] further motivates TRUST to show that interfacing standalone safe Rust and C/C++ hardened against control flow hijacking can reintroduce the vulnerability. PKRU-Safe [26] uses MPK to isolate Rust from untrusted code as TRUST does. PKRU-safe associates each allocation site with a unique ID and performs profiling with the developer-provided inputs to classify the allocation sites into safe and unsafe ones. Unfortunately, this approach suffers from two drawbacks that TRUST overcomes with function cloning (see §5.2) and static points-to analysis (see §5.1). In Rust, a handful of smart pointer APIs produces almost all pointers, causing context-insensitive analysis to associate all such pointers with the same allocation site that has the same ID. TRUST overcomes this problem by using sound and context-sensitive static analysis to find safe allocation sites and clone functions to associate each clone with a distinct ID.

8 Conclusion

To our knowledge, TRUST is the first attempt to automatically quarantine from safe Rust blocks all major sources of the untrusted code, including unsafe blocks and external libraries, that undermine Rust's strong security guarantees. It is the first compiler framework that addresses the challenges of automatically identifying and protecting safe objects in a given Rust program. Thanks to this automation, unlike existing mechanisms, TRUST protects safe objects both on stack and heap without human intervention from unsafe blocks and external libraries written in unknown languages. In addition, TRUST comes with an elaborated instrumentation strategy and runtime libraries that help us to attain lower performance overhead (7.55%) than state-of-the-art (36.39%). It also defeats the counterfeit stack attack via our carefully designed gates to the external libraries. In short, TRUST is an automatic mechanism that can quarantine inclusively any untrusted code linked and integrated into safe Rust.

Acknowledgment

This research was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government, Ministry of Science and ICT(MSIT) (NRF-2020R1A2B5B03095204 and NRF-2022R1F1A1076100), the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2022, and Inter-University Semiconductor Research Center (ISRC). Also, the research was supported by the MSIT, Korea, under the ITRC(Information Technology Research Center) support program(IITP-2022-2020-0-01602) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation). Finally, this work was supported by the IITP grant funded by the Korea government(MSIT) (No.2021-0-00724, RISC-V based Secure CPU Architecture Design for Embedded System Malware Detection and Response, and No.2021-0-01817, Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters,), and Samsung Electronics Co., Ltd.

References

[1] Build reliable network applications without compromising speed. https://tokio.rs. Accessed: 2022-01-18.

[2] Cve-2018-1000657 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-1000657. Accessed: 2022-01-18.

[3] Cve-2018-1000810 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-1000810. Accessed: 2022-01-18.

[4] Cve-2019-15546 detail. https://nvd.nist.gov/vuln/detail/CVE-2019-15546. Accessed: 2022-01-18.

[5] Cve-2021-28028 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-28028. Accessed: 2022-01-18.

[6] Cve-2021-28030 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-28030. Accessed: 2022-01-18.

[7] Cve-2021-28879 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-28879. Accessed: 2022-01-18.

[8] Cve-2021-29939 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-29939. Accessed: 2022-01-18.

[9] Cve-2021-45707 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-45707. Accessed: 2022-01-18.

[10] hyper: Fast and safe http for the rust language. https://hyper.rs. Accessed: 2022-01-18.

[11] Memory safety issue in stackvec::extend. https://github.com/Alexhuszagh/rust-stackvector/issues/2. Accessed: 2022-01-18.

[12] parasol-aser/xrust. https://github.com/parasol-aser/XRust. Accessed: 2022-01-18.

[13] Smart pointers. https://doc.rust-lang.org/book/ch15-00-smart-pointers.html. Accessed: 2022-01-19.

[14] Snappy, a fast compressor/decompressor. =https://github.com/google/snappy.

[15] Hussain M. J. Almohri and David Evans. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 248–255, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, October 2021.

[17] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 158–168, New York, NY, USA, 2006. Association for Computing Machinery.

[18] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 10:368–379, November 2013.

[19] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, 2016.

[20] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. Pku pitfalls: Attacks on pku-based memory isolation systems. In *USENIX Security Symposium*, 2020.

[21] Liang Deng, Qingkai Zeng, and Yao Liu. Isboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP International Information Security Conference*, 2015.

[22] Mark A. Finlayson. *Intel® 64 and IA-32 architectures software developer's manual*, 2020.

[23] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 517–528, New York, NY, USA, 2016. Association for Computing Machinery.

[24] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[25] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[26] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 132–148, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *EuroSys*, April 2017.

[28] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, page 51–57, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[30] Daan Leijen, Benjamin G. Zorn, and Leonardo Mendonça de Moura. Mimalloc: Free list sharding in action. In *APLAS*, 2019.

[31] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with xrust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 234–245, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[33] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *NDSS*, 01 2022.

[34] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. 01 2020.

[35] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, pages 31–40, 2010.

[36] Gene Novark and Emery D. Berger. Dieharder: Securing the heap. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, page 12, USA, 2011. USENIX Association.

[37] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[38] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, Boston, MA, August 2022. USENIX Association.

[39] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient In-Process isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[41] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403, 2017.

[42] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 265–266, New York, NY, USA, 2016. Association for Computing Machinery.

[43] Yulei Sui and Jingling Xue. Demand-driven pointer analysis with strong updates via value-flow refinement. *CoRR*, abs/1701.05650, 2017.

[44] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery.

[45] The Rust team. *The Rust programming language*, 2017.

[46] John Toman, Stuart Pernsteiner, and Emina Torlak. Crust: A bounded verifier for rust (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 75–80, 2015.

[47] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[48] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by)pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 266–282, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 203–216, New York, NY, USA, 1993. Association for Computing Machinery.

[50] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2333–2350, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 299–308, New York, NY, USA, 2012. Association for Computing Machinery.

[52] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[53] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[54] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. pages 631–644, 04 2019.

[55] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. Armor: Fully verified software fault isolation. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 289–298, 2011.

A Absolute Numbers for the Experimental Result

Table 6: Normalized execution time of TRUST tested with large benchmarks (Figure 12).

Benchmark	Baseline(/iter)	TRUST (/iter)
hyper-body	1.16μs	1.2μs(1.11×)
hyper-connect	26.85μs	31.24μs(1.16×)
hyper-end_to_end	11.45ms	11.58ms(1.01×)
hyper-pipeline	18.73μs	30.42μs(1.62×)
hyper-server	11.04ms	16.15ms(1.46×)
tokio-rwlock	2.37μs	2.49μs(1.05×)
tokio-semaphore	2.33μs	3.47μs(1.49×)
tokio-mpsc	0.40ms	1.12ms(2.54×)
tokio-signal	8.88μs	13.71μs(1.54×)
tokio-fs	2.37ms	2.37ms(1.00×)

Table 7: Performance overhead of 11 widely used crates (Figure 9).

Benchmark	Baseline	TRust-jemalloc	TRust-mimalloc	XRust
Base64	91.3ms/iter	90.7ms/iter(0.99×)	84.0ms/iter(0.92×)	0.1s/iter(1.59×)
Bytes	71.2μs/iter	93.4μs/iter(1.31×)	91.1μs/iter(1.28×)	80.0μs/iter(1.12×)
Byteorder	2.2ms/iter	2.0ms/iter(0.92×)	2.0ms/iter(0.92×)	2.2ms/iter(1.00×)
Json	1.4ms/iter	1.4ms/iter(1.00×)	1.3ms/iter(0.92×)	1.5ms/iter(1.04×)
Image	24.1ms/iter	24.1ms/iter(1.00×)	25.0ms/iter(1.04×)	24.6ms/iter(1.02×)
Regex	0.2s/iter	0.2s/iter(1.02×)	0.2s/iter(1.04×)	0.2s/iter(1.01×)
Vec	28.3μs/iter	28.4μs/iter(1.00×)	28.0μs/iter(0.99×)	58.1μs/iter(2.05×)
String	34μs/iter	36μs/iter(1.06×)	35.7μs/iter(1.04×)	70.7μs/iter(2.06×)
Linked-list	1.0μs/iter	1.9μs/iter(1.94×)	1.8μs/iter(1.84×)	1.2μs/iter(1.21×)
Vec-deque	1.9μs/iter	2.0μs/iter(1.02×)	2.0μs/iter(1.03×)	2.2μs/iter(1.16×)
Btree	0.4ms/iter	0.4ms/iter(1.02×)	0.4ms/iter(1.04×)	0.5ms/iter(1.11×)
GeoMean	0.4ms/iter	0.5ms/iter(1.09×)	0.5ms/iter(1.07×)	0.5ms/iter(1.26×)

Table 8: Performance overhead of snappy (Figure 10).

Size (Bytes)	Compress				Uncompress			
	Baseline	TRust	Baseline	Sandcrust	Baseline	TRust	Baseline	Sandcrust
256	0.1ms	0.2ms(1.58×)	0.6ms	5.5ms(9.17×)	0.1ms	0.2ms(2.16×)	0.2ms	10ms(50.00×)
1K	0.3ms	0.3ms(0.86×)	1.5ms	7.0ms(4.67×)	0.1ms	0.2ms(2.01×)	0.5ms	25.0ms(50.00×)
4K	0.4ms	0.5ms(1.14×)	6.0ms	10.5ms(1.75×)	0.1ms	0.2ms(1.84×)	2.0ms	90.0ms(45.00×)
16K	0.4ms	0.5ms(1.09×)	20.0ms	30.0ms(1.50×)	0.1ms	0.3ms(1.91×)	7.0ms	0.3s(42.86×)
64K	2.3ms	2.6ms(1.13×)	90.0ms	0.2s(1.67×)	1.4ms	1.5ms(1.04×)	40.0ms	0.7s(17.50×)
256K	11.1ms	11.8ms(1.06×)	0.3s	0.4s(1.33×)	8.0ms	8.2ms(1.03×)	0.2s	2.5s(16.67×)
1G	44.0ms	43.9ms(1.00×)	1.5s	2.0s(1.33×)	31.9ms	31.6ms(0.99×)	0.7s	10.0s(14.29×)
4G	0.2s	0.2s(1.03×)	6.0s	9.0s(1.50×)	0.1s	0.1s(1.03×)	3.0s	40.0s(13.33×)
16G	2.2s	2.2s(1.03×)	25.0s	35.0s(1.40×)	1.3s	1.2s(0.90×)	10.0s	0.2ks(20.00×)
GeoMean	5.4ms	5.8ms(1.09×)	95.4ms	0.2s(2.07×)	2.8ms	3.7ms(1.35×)	38.3ms	1.0s(25.96×)

Table 9: Memory impact of heap allocator (Figure 11).

Benchmark	Baseline	Baseline(mimalloc)	TRust(jemalloc)	TRust(mimalloc)	XRust
Base64	82.4MB	0.1GB(1.48×)	83.7MB(1.02×)	83.9MB(1.02×)	82.5MB(1.00×)
Bytes	3.2MB	3.6MB(1.10×)	4.5MB(1.40×)	3.5MB(1.07×)	3.4MB(1.04×)
Byteorder	4.7MB	5.2MB(1.10×)	6.0MB(1.29×)	5.3MB(1.12×)	4.7MB(1.00×)
Json	0.7GB	0.7GB(1.00×)	2.0GB(2.95×)	1.1GB(1.67×)	0.7GB(1.00×)
Image	5.2MB	6.3MB(1.22×)	6.3MB(1.21×)	7.0MB(1.33×)	5.1MB(0.99×)
Regex	0.2GB	0.2GB(1.03×)	0.2GB(1.01×)	0.2GB(1.01×)	0.2GB(1.00×)
Vec	3.2MB	3.5MB(1.09×)	4.6MB(1.43×)	3.5MB(1.10×)	3.3MB(1.04×)
String	3.2MB	3.6MB(1.11×)	4.7MB(1.46×)	3.6MB(1.12×)	3.3MB(1.02×)
Linked-list	1.4GB	1.6GB(1.15×)	1.4GB(1.04×)	1.3GB(0.95×)	2.5GB(1.83×)
Vec-deque	3.1MB	3.4MB(1.11×)	4.5MB(1.45×)	3.4MB(1.10×)	3.2MB(1.04×)
Btree	4.9MB	5.2MB(1.06×)	6.2MB(1.26×)	5.4MB(1.11×)	5.1MB(1.04×)
GeoMean	19.5MB	22.0MB(1.13×)	26.3MB(1.35×)	22.1MB(1.13×)	20.1MB(1.07×)

Table 10: Ratio of safe stack/heap allocation (Figure 13 and Figure 14).

Benchmark	TRust Stack		TRust Heap		XRust Heap	
	Ratio	Safe/Total	Ratio	Safe/Total	Ratio	Safe/Total
Base64	0.00	(18.7K/10.5M)	0.45	(3.4M/ 8.5M)	0.44	(17.6M/31.1M)
Bytes	0.67	(0.9G/ 1.4G)	0.67	(2.5M/ 3.7M)	0.40	(32.6M/48.6M)
Byteorder	1.00	(7.1K/ 7.1K)	1.00	(3.2M/ 3.2M)	1.00	(14.0K/14.0K)
Json	0.98	(0.2G/ 0.2G)	0.00	(1.6K/18.6M)	0.95	(18.5M/19.4M)
Image	0.54	(15.6M/29.1M)	0.60	(0.4M/ 0.6M)	0.92	(19.1M/20.7M)
Regex	0.51	(2.4M/ 4.7M)	1.00	(5.2G/ 5.2G)	1.00	(4.3M/ 4.3M)
Vec	0.88	(0.1M/ 0.1M)	0.36	(35.7M/98.1M)	0.81	(0.3G/ 0.4G)
String	1.00	(5.0K/ 5.0K)	1.00	(0.1G/ 0.1G)	1.00	(0.2G/ 0.2G)
Linked-list	1.00	(1.3K/ 1.3K)	1.00	(0.2G/ 0.2G)	1.00	(0.3G/ 0.3G)
Vec-deque	1.00	(1.4K/ 1.4K)	1.00	(12.5M/12.5M)	1.00	(13.2M/13.2M)
Btree	1.00	(0.4G/ 0.4G)	0.37	(7.9K/21.4K)	1.00	(25.9K/25.9K)