Cross-Language Attacks

Samuel Mergendahl
MIT Lincoln Laboratory
samuel.mergendahl@ll.mit.edu

Nathan Burow
MIT Lincoln Laboratory
nathan.burow@ll.mit.edu

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

***Abstract*—Memory corruption attacks against unsafe programming languages like C/C++ have been a major threat to computer systems for multiple decades. Various sanitizers and runtime exploit mitigation techniques have been shown to only provide partial protection at best. Recently developed 'safe' programming languages such as Rust and Go hold the promise to change this paradigm by preventing memory corruption bugs using a strong type system and proper compile-time and runtime checks. Gradual deployment of these languages has been touted as a way of improving the security of existing applications before entire applications can be developed in safe languages. This is notable in popular applications such as Firefox and Tor. In this paper, we systematically analyze the security of multi-language applications. We show that because language safety checks in safe languages and exploit mitigation techniques applied to unsafe languages (*e.g.,* Control-Flow Integrity) break different stages of an exploit to prevent control hijacking attacks, an attacker can carefully maneuver between the languages to mount a successful attack. In essence, we illustrate that the incompatible set of assumptions made in various languages enables attacks that are not possible in each language alone. We study different variants of these attacks and analyze Firefox to illustrate the feasibility and extent of this problem. Our findings show that gradual deployment of safe programming languages, if not done with extreme care, can indeed be detrimental to security.**

I. Introduction

A new generation of modern, safe programming languages have been developed that perform security checks natively [63], motivated partly by the limitations of defenses applied to unsafe languages (like C/C++) [27], [41], [46], [52], [54], [114]. Rust [71] and Go [72] are two such languages that prevent the introduction of memory corruption bugs by virtue of having a strong type system and by performing proper compile-time and runtime checks. For example, Rust's type system prevents arbitrary casting, performs compile-time ownership checks to prevent temporal memory safety bugs, and enforces compile-time bounds checks on static data combined with runtime bounds checks on dynamic data to prevent spatial memory corruption bugs [71]. As another example, Go has a garbage collector to provide temporal memory safety [72]. While these languages provide keywords to ignore the safety checks of the language when necessary (*e.g.,* the `unsafe` keyword in Rust is used to interact with low-level hardware devices), within the confines of the safe code, the applications written in these languages are considered generally safe. In fact, these languages have been touted as the 'best chance' to develop safe systems [57] and their gradual deployment is underway in multiple popular applications and code bases. These include, but are not limited to: Firefox [47], Tor [14], Microsoft Windows operating system [117], Google Fuchsia OS [48], and multiple flavors of Linux [8], [11] developed in part in Rust, as well as, Docker [5], Kubernetes [7], CockroachDB [2], and BoltDB [3] developed in part in Go. This has resulted in the deployment of multi-language applications, in which two or more languages are used in development.

In this paper, we analyze the security of multi-language applications. Since unsafe languages without additional protections are trivially vulnerable to memory corruption attacks, we specifically focus on the case where some protection is applied to the unsafe side (*e.g.,* CFI for C/C++) and the safe side does not contain `unsafe` code. In these cases, the incremental development of parts of the application in the safe programming language is performed to 'enhance' its security. For example, the Servo CSS style calculation in Firefox [12], Dogear (a bookmark merger for Sync in Firefox) [6], the MP4 metadata parser in Firefox [9], and the neqo QUIC implementation in Firefox [10] are all implemented in Rust, while many other parts of Firefox are in C and C++, among other languages. We build a model of how various runtime exploit mitigation checks and language safety checks attempt to break different stages of an exploit. We further illustrate that these checks create an incompatible set of assumptions on each side. Leveraging these incompatibilities in the safety checks performed, we show that an attacker can maneuver between the languages in a way that allows the exploit to succeed without violating the safety checks on either side. In other words, the introduction of a safe language creates a conflicting set of assumptions that indeed weakens the security of both sides. We illustrate that a new vector of attack, Cross-Language Attacks (CLA), becomes possible in such settings which results in control-flow hijacks that are otherwise prevented on each language individually.

We study different variants of CLA with concrete code samples based on the stages of an exploit that are broken by the security checks. Our examples focus on Rust for simplicity of exposition, but generalize to Go and other language combinations (see discussion in Section VII-A and Section VII-B and our Go code samples in Appendix A). Moreover, to

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24078
www.ndss-symposium.org

illustrate the extent of this problem, we perform automated analysis on Firefox, which we believe is representative of large, commonly-used code bases and quantitatively assess the conditions that make CLA possible. Additionally, we make our analysis and concrete code examples available online[1].

Our findings illustrate that, incremental deployment of safe languages, if not done with extreme care, can indeed be detrimental to security. An attacker can leverage the incompatible set of assumptions made by various languages to craft CLA where typical control-flow hijacking is prevented by each language individually.

The contributions of this paper are as follows:

- We study the security of multi-language applications. We develop a model to reason about the stages of an exploit broken by each type of security check and we illustrate that different languages have incompatible assumptions about these stages.
- We illustrate that by leveraging these conflicting assumptions, an attacker can maneuver between languages in a way that allows control-flow hijacking where none is possible in individual languages alone.
- We demonstrate different variants of CLA that use discrepancies in different exploitation stages. We demonstrate these attacks on Rust, and for both intended and unintended interactions. We further show that not only the overall CLA is made possible by multiple languages, but also some of the underlying vulnerabilities arise from the multi-language setting.
- We automatically analyze Firefox, a large, popular, and open source code base to highlight the prevalence of opportunities for CLA. Our findings illustrate that incremental deployment of safe languages, if not done with proper care, can indeed be detrimental to security.

II. Background

The literature in the area of memory corruption attacks and defenses is vast. A solid treatment of this area is beyond the scope of this paper. In this section, we provide a quick background and focus on areas that are needed to understand the rest of the paper. We refer interested readers to surveys and systematization of knowledge papers in this area for a more complete treatment of the subject [26], [66], [108], [114].

Memory corruption attacks have posed a major threat to computer systems for decades [84], [114]. The complexity of these attacks have increased over the years from simple stack-based code injection attacks [84] to various forms of code reuse attacks [101]. The underlying root cause of memory corruption attacks is the unsafety of programming languages like C/C++ that delegate security checks to the developer. Developer mistakes thus result in the introduction of spatial (*e.g.,* buffer overflows) and temporal (*e.g.,* use-after-free) memory corruption bugs that can be exploited by an attacker.

In response, various offline analysis tools (*i.e.,* sanitizers) [108] and runtime exploit mitigation techniques [26], [66], [114] have been developed to find these bugs prior to deployment and prevent their exploitation while deployed, respectively. Runtime exploit mitigation techniques can be further categorized into enforcement-based techniques [114] and randomization-based ones [66], [80]. Control-flow integrity (CFI) [25] is an example of the former, while memory randomization [86] is an example of the latter. However, these techniques have been shown to only provide partial protection at best. Sanitizers suffer from coverage limitations, often resulting in many missed bugs [108]. In addition, enforcement-based exploit mitigation techniques are shown to provide relaxed-enough policies that allow an attacker to mount a successful attack without violating their policies [27], [41], [52], while randomization-based techniques are shown to be vulnerable to various forms of information-leakage attacks [40], [99], [110]. Perhaps the ultimate indicator of the limitations of securing unsafe languages is the prevalence of memory corruption bugs in modern systems, despite attempts to catch or mitigate them during development and while deployed. According to multiple studies, memory corruption bugs account for a large fraction of vulnerabilities (up to 70%) today [73].

The limitations of protections applied to unsafe programming languages have motivated a new generation of programming languages that provide safety properties natively [81], [82]. Rust and Go are two such languages. We briefly describe how Rust and Go provide memory safety below.

A. Rust

Rust [71] is a multi-paradigm programming language that provides strong performance and safety properties. Rust has a C-like syntax, but a strong type system combined with compile-time and runtime checks to prevent large classes of bugs, such as memory corruption and concurrency bugs. Rust's small language runtime makes it appropriate for systems programming, which has resulted in multiple operating systems and low-level code being developed using it [8], [11], [14], [47], [48], [117].

Rust has a strong type system and enforces both spatial and temporal memory safety [63]. For spatial safety, Rust has a two pronged approach. For statically-sized objects, it performs a compile-time size check to avoid out-of-bound accesses. For dynamically-sized objects or for static objects with unknown indices (*e.g.,* an array with a variable index), Rust inserts proper instructions in the binary to perform bounds checking at runtime. Rust also has a strong type system that prevents raw pointers and unsafe casting.

For temporal memory safety, Rust's solution is more innovative and at the same time restrictive. Rust has a notion of *ownership*. Each value in Rust has a variable that is its owner. Only one owner of a value can exist at a time and when the owner goes out of scope, the value is destroyed. To allow values to be passed between different parts of a code, Rust uses borrowing, which is a temporary transfer of ownership. As a generalization of this principle, Rust only allows one 'mutable reference' (*i.e.,* 'pointer' in other languages) or multiple immutable references to exist to an object, but not both. The advantage of this design is that when a value is destroyed, Rust can easily nullify any reference to it without the need for heavy-weight garbage collection. This, in addition to other factors, make Rust appropriate for

[1] https://github.com/mit-ll/Cross-Language-Attacks

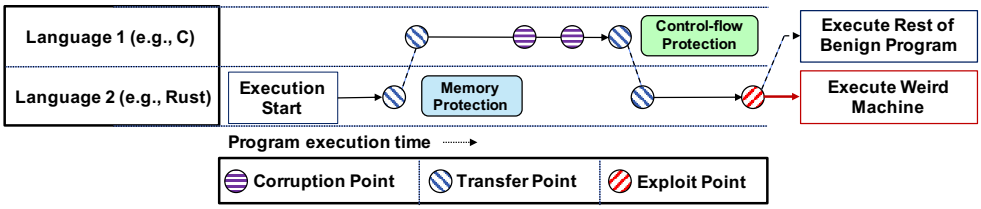


Fig. 1: Cross-Language Attacks (CLA) transfer back and forth between languages to circumvent deployed defenses.

system programming. By the same token, this rule is also too restrictive at times. For example, in a doubly-linked list, there needs to be two mutable references to each value at a time.

Rust's mechanism for breaking out of these rules is the `unsafe` keyword. Code enclosed in an `unsafe block` (herein simply referred to as 'unsafe Rust') can dereference raw pointers and avoid ownership rules. Unsafe Rust is necessary when interacting with low-level devices (*e.g.,* writing to a memory-mapped I/O device). It can also be used to develop data structures that are internally unsafe (*e.g.,* doubly-linked lists), but only expose safe interfaces in what is known as *interior mutability* [71]. Many such data structures are formally shown to be indeed safe [58]. The dangers of unsafe Rust have been acknowledged in the literature [65], [69], and independently investigated [18]. Here, we primarily focus on novel vulnerabilities from mixing *safe* Rust with unsafe languages.

Rust allows interactions with other languages through its foreign function interface (FFI). FFI is inherently unsafe in Rust, and allows the exchange of arbitrary data, including pointers, across the language boundary. FFI allows the exchange of arbitrary data (including raw pointers) between Rust and other languages, most notably C. Rust has additional rules to make FFI less dangerous, for example, dynamically-sized types cannot be used for FFI. Having said that, the boundary between Rust and a language like C is, by its very nature, unsafe. Indeed, a call through Rust FFI requires the `unsafe` keyword.

We refer to transfers of control flow between languages in a multi-language application through FFI as *intended* interactions. *Unintended* interactions are also possible as applications within the multi-language application share an address space. While intended interactions have been the subject of some prior work [77], and similarly, some previous work encourages sandboxing safe language memory from unsafe languages [91], we believe the relationship between intended/unintended interactions and the preservation of language threat models in multi-language application have to date been under investigated by the community.

B. Go

Go [72] is a multi-paradigm programming language that is statically-typed. It provides spatial safety primarily through runtime bounds checks. It also performs various optimizations at compile-time to eliminate unnecessary bounds checks such as redundant checks inside a loop.

In order to provide temporal safety, Go deploys garbage collection (GC). Go does not have any limitation on the number or usage of pointers, which allows the development of complex data structures. On the other hand, the down side of GC is latency and CPU utilization, which can be substantial (∼25% CPU utilization) depending on the code [90]. This also makes Go's language runtime significantly more complex than that of Rust. By the same token, Go binaries are generally larger than those of Rust.

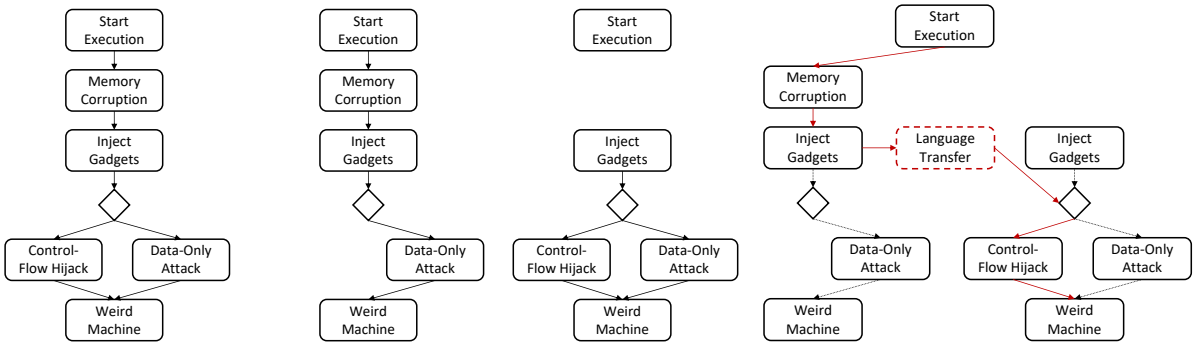
Similar to Rust, Go can also interact with other languages. For example, CGo [4] allows calling C code from Go. This includes passing Go pointers to C, which can indeed become dangling [94]. Dangerously, Go also hides the inclusion of C code during compilation without warning.

III. CLA Model

We build on existing work [114] presenting high-level threat models for software security, and extend these models to hardened and multi-language applications. In particular, we show that the threat model for a multi-language application is the union of the threat models of the constituent languages. In graphical terms, creating a multi-language application threat model involves adding edges from each node in the threat model to a new "language transfer" node. This can lead to multi-language applications being weaker than their constituent parts due to CLA, a concerning negative synergy.

At a high-level, a CLA is illustrated in Figure 1. The CLA starts its execution in one language (in this example, Rust). Because of the memory safety checks in the safe language, corruption is not possible, so the CLA proceeds by `transferring` to the unsafe language (in this example, C) for the actual memory corruption. However, because of the protections applied to the unsafe language (in this example, CFI), control-flow hijacking is not possible there, so the CLA transfers back to the safe language to execute the weird machine [103]. The unsafe language assumes that the hardening (*e.g.,* CFI) prevents the hijacking of control and the safe language assumes that the initial corruption is not possible, so it does not check the transfer of control to a weird machine. Consequently, by carefully maneuvering between the languages, the CLA can succeed in a multi-language application even when it is not possible in individual languages separately. We describe the details of such attack further in the upcoming sections.

In this section, we first discuss the threat models for prevalent programming languages, focusing on compiled languages. We then present a novel graph-based analysis of the threat models that demonstrates that multi-language application have the pair-wise weaknesses of their constituent languages. The composition of language threat models is illustrated in Figure 2.



(a) C threat model (b) C + CFI threat model (c) Rust/Go threat model (d) CLA threat model

Fig. 2: Language Threat Models

A. Single-Language Application Threat Models

Figure 2a illustrates the basic chain of events in a memory corruption based software attack, and is modeled off C with only DEP [15], Stack Canaries [119], and ASLR [86] protections (*i.e.,* standard C with no added security). An attacker steers execution towards a memory corruption, which is used to modify the application's memory layout per an attacker's specifications (*i.e.,* inject gadgets). These gadgets are then used by the attacker to assume control over the application, either directly by overwriting a code pointer in a control-flow hijack or more subtly and indirectly in DOP [52] attacks. Once the attacker assumes control, they execute the weird machine [103] that their memory corruption set up, and achieve their goals. Attacks thus have four essential phases: i) memory corruption, ii) gadget injection, iii) control-flow assumption, and iv) weird machine execution. To stop an attack, it is sufficient for a defender to disrupt any of these steps, though in practice defenses have focused on steps i and iii [114].

Figure 2b shows the updated threat model for C with (ideal) CFI [25] hardening. Note that the "Control-Flow Hijack" node has been deleted, which is the result of a perfect pointer protection defense (in practice, however, CFI falls short of this standard [27], [41], [42]). Removing this node forces attackers to rely on DOP [52] attacks to execute their weird machines, significantly raising the bar for attackers.

Memory safety, as provided by modern languages such as Rust and Go, offers a strong defense by removing the "memory corruption" node, see Figure 2c. Removing the root cause of an attack removes all of the downstream variants, but experience has shown it must be designed into the language; decades of attempts to retrofit memory safety into C [46], [108], [114] have essentially resulted in only partial protection at best.

B. Multi-Language Application Threat Models

Given the single-language application threat models in Figure 2a, Figure 2b, and Figure 2c, it is important to correctly compose these underlying threat models for CLA. Multi-language applications introduce a new primitive to the threat model: *Language Transfer* nodes. Language transfers occur when an application deliberately interacts with a component in another language (*e.g.,* through FFI).

Conservatively, each node in the constituent language threat models must connect to the Language Transfer node, as there is no way of knowing when language transfers occur in an application. We cannot say, for example, that all language transfers happen before any possible memory corruption. Consequently, the threat models for multi-language application are fully connected and attacks eliminated by hardening one language may become possible when composing languages.

Figure 2d illustrates how single-language application threat models compose for a Rust, *c.f.,* Figure 2c, and CFI hardened C, *c.f.,* Figure 2b, multi-language application enabling an attack that is not possible in either component. CFI hardened C applications prevent control-flow hijacking by validating code pointers before they are used. Rust applications prevent the same attack by enforcing memory safety. Note, however, that CFI hardened C is not memory safe, and Rust does validate code pointers before they are used, as it assumes memory safety. Consequently, an attacker can use a memory corruption in C and a non-validated indirect call site in Rust to create a control-flow hijacking attack in a Rust-C multi-language application.

The fundamental problem here is a mismatch of assumptions in the individual language constituents of a multi-language application. Multi-language applications have the security of their weakest constituent language. While we have illustrated the issue here with a classic control-flow hijacking attack, the problem is much deeper than that. The weakest link principle holds for any element of an application's threat model that varies across languages. For instance, if Rust were to introduce code signing and validation to mitigate supply-chain attacks and C libraries did not, then a multi-language application composed of those two languages would remain completely vulnerable to supply chain attacks.

The most insidious case of the multi-language application threat model composition is when both constituent languages have eliminated a threat, but have done so using different assumptions. The multi-language application then undermines both sets of assumptions, resulting in the combination of two "safe" languages itself being unsafe. Even for the common scenario of hardening a legacy codebase, such as a C codebase, with a new component written in a safe programming language, such as Rust, this "hardening" can actually end up *weakening* the application's security.

C. CLA Attack Construction

Given the CLA threat model, we next discuss more concretely the construction of an end-to-end CLA attack. Then, in the next section, we focus on components of such attacks that are unique to the CLA scenario, and introduce new attack primitives. To this end, we next present a more detailed graphical description of CLA in Figure 3. Each node in the graph represents a potential step in the attack, with arrows as indication of possible sequences of steps. A successful attack is a traversal from `Execution Start` to `Weird Machine Execution`. Any such traversal that contains the `Language Transfer` node is a CLA.

As in Figure 2d, we encode the defensive guarantees of Rust and CFI hardened C in Figure 3. Rust's type system, and in particular its borrow checker, lifetimes, and dynamic bounds checks, provide memory safety which defends the `Memory Corruption` node (shaded blue). The expansion of the `Memory Corruption` node for Rust shows that the initial steps of memory corruption, the temporal or spatial vulnerabilities, are removed by Rust. Similarly for CFI hardened C, the `Weird Machine Execution` node is defended (shaded green). As the expansion shows, CFI prevents an indirect call / jump from using an arbitrary attacker controlled code-pointer[2]. Combined with a shadow stack [26] to protect returns, hardened C on its own is also largely immune to control hijacking attacks, though data-only attacks [53] remain a threat.

The red nodes in Figure 3 are a concrete instantiation of a CLA attack for illustration, following the concrete attack presented by Papaevripides et al. [85]. The attacker first steers execution towards a known memory safety bug, ①. This leads to the attacker obtaining a "write what where" vulnerability, ②, and using it to inject gadgets, ③. The attacker then steers execution towards a language transition, ④–⑧, and finally, the attacker uses the corrupted code pointer to launch their code reuse attack, ⑨. The only complication here from a classic code reuse attack is the need to find a language transfer point. At the binary level where attacks are constructed, however, this problem is simplified to finding an unprotected indirect call for the attack to target. With such a point and a vulnerability, existing techniques such as Block Oriented Programming [54] can successfully construct attacks.

The simplicity of constructing such attacks at the binary level makes CLA significantly more dangerous. Namely, such attacks can be constructed unknowingly by adversaries looking for classic attack patterns, as opposed to COOP [97] or DOP [52] attacks that require new primitives. We next discuss numerous CLA variants, including first, using old vulnerabilities brought back to life by CLA (that we denote as CLA using Revenant Vulnerabilities found in Section IV), and second, using new vulnerabilities that only exist in multi-language applications (that we denote as CLA using Multi-Language-Specific Vulnerabilities found in Section V).

IV. CLA using Revenant Vulnerabilities

We present a series of attack variants demonstrating that vulnerabilities typically mitigated by safety check/hardening techniques re-emerge as revenant vulnerabilities using CLA in multi-language applications. We focus our exposition on Rust-C/C++ applications, and show that key defensive primitives either built into or commonly applied to Rust and C/C++ respectively are completely bypassed by CLA. An overview of the attacks we present in this section is contained in Table I. We show that the spatial and temporal memory safety defenses of Rust are bypassed by CLA, as are Shadow Stacks [26] and CFI [25] for C/C++. We facilitate our discussion with a series code examples—found in Figures 6 to 10—for exposition. While our examples focus on Rust for simplicity of exposition, we point the reader to Appendix A for an illustration of similar examples in Go. Moreover, we typically use C and C++ interchangeably throughout the following sections.

A. Overview

A key feature of Rust is memory safety, which rests on two pillars: Rust's expressive type system and its automatically inserted, dynamic checks. The type system can prove many accesses to be spatially safe at compile time, but some accesses require simple checks at runtime against a constant size bound (*e.g.,* random access into a fixed size array). However, for objects whose size is not known at compile time, such as `vectors`, Rust stores the bounds information in memory, and performs bounds checks against it. All indexes into objects are unsigned, meaning Rust only has to perform upper bounds checks and not lower.[3] All of this machinery is for nought in multi-language applications, however, as arbitrary write vulnerabilities in C/C++ can effect *any* memory in the shared application's address space. Such attacks are simplified with a pointer to Rust memory, such as a Rust heap object that contains a function pointer or the Rust stack, but by no means require such a pointer. An example of such an attack is in Figure 4.

Rust's temporal memory safety relies on the ownership model of its type system, which is used for automatic memory management. While programmers can force heap allocations, they are usually oblivious to whether a variable is stack or heap allocated, and deallocation is handled automatically when the variable goes out of scope. However, there is nothing preventing double frees as a result of FFI—as we illustrate below—or preventing programmer error from causing a Use-after-Free (UaF) as a result of FFI. Given that FFI requires unsafe code, responsibility for memory management returns to the programmer, reintroducing such errors.

CFI is entering widespread usage in C/C++ applications, and is designed to provided partial memory safety by protecting the integrity of code pointers. In combination with Shadow Stacks, CFI offers the best combination of strength and performance among runtime defenses to date. As discussed previously, *c.f.,* Figure 2, Rust does not use CFI as it provides full memory safety, rendering partial memory safety redundant. Consequently, an arbitrary write vulnerability in C/C++ can corrupt a code pointer used in Rust, bypassing CFI verification or shadow stack protection. See Figure 5 for an example.

[2] An attacker can still redirect within the set of allowed targets, which has been shown to be sufficient for attacks [27].

[3] It is interesting to note that these three categories of checks map nicely to the CCured [79] type system that is now 16 years old, highlighting how long and winding the road to practical memory safety has been.

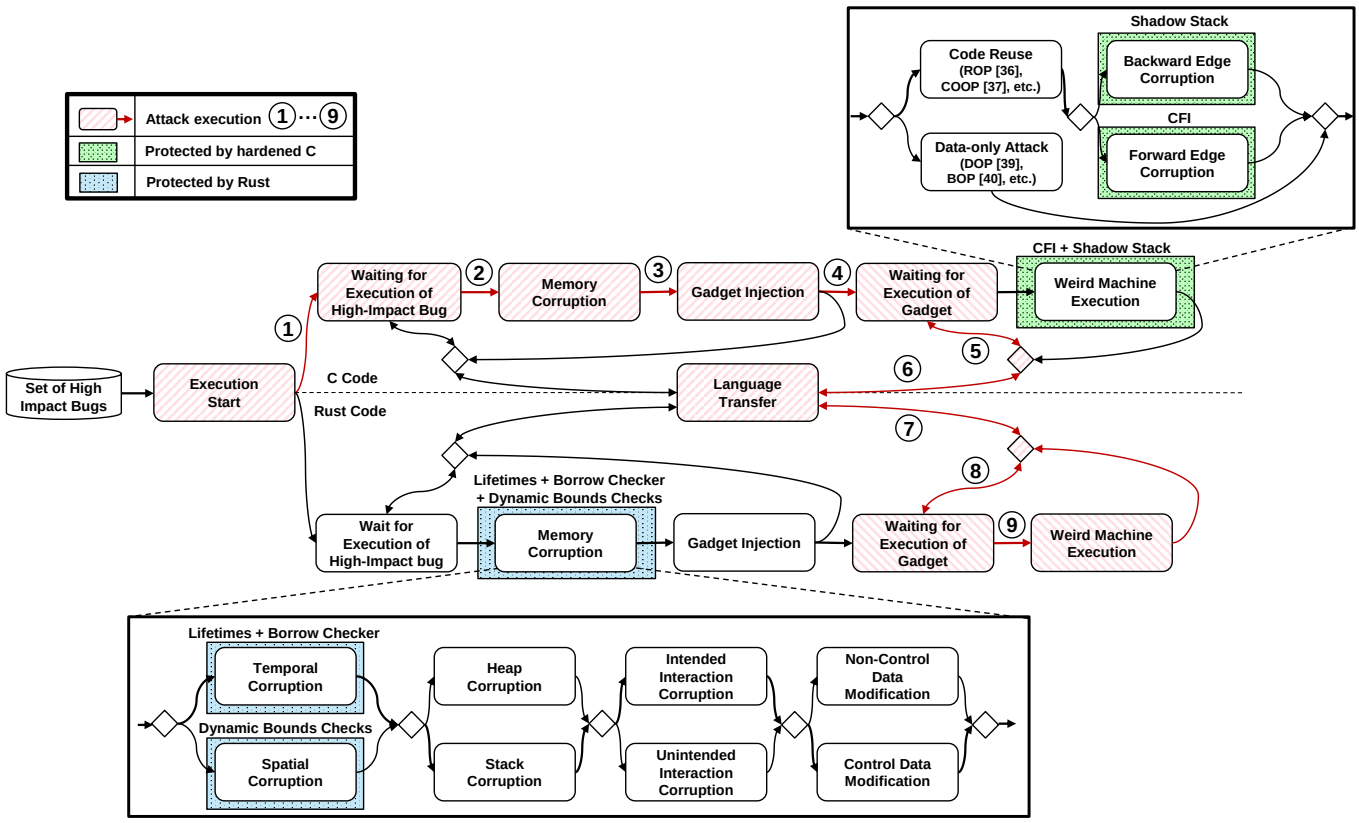


Fig. 3: Our baseline attack variant on a program with Rust (protected by its lifetimes, borrow checker, and dynamic bounds checks) and C (protected by stack canaries and CFI) colored over a graphical description of all Cross-Language Attacks (CLA).

TABLE I: CLA Variants using Revenant Vulnerabilities

Targeted Language	Bypassed Defense	Memory Corruption Used		Weird Machine Execution Origin	
		Spatial	Temporal	Forward Edge Corruption	Backward Edge Corruption
Rust	Bounds Checks	✔		✔	✔
	Lifetimes		✔	✔	
C++	Shadow Stack	✔			✔
	CFI	✔	✔	✔	

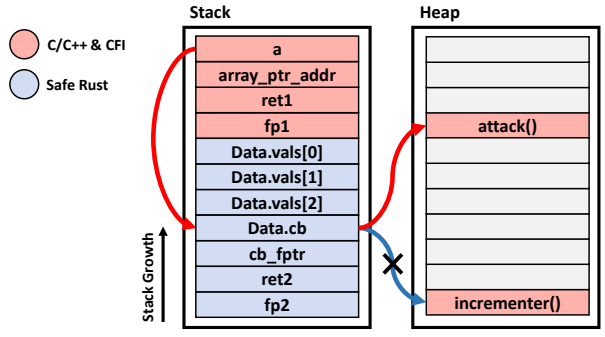


Fig. 4: C/C++ is not subject to the Rust type system, so it can dereference a pointer out-of-bound to a Rust function pointer and make it point to an attacker chosen gadget.

Note that all the examples contained below are simplified for discussion here. One can find the full working versions online[4].

[4]https://github.com/mit-ll/Cross-Language-Attacks

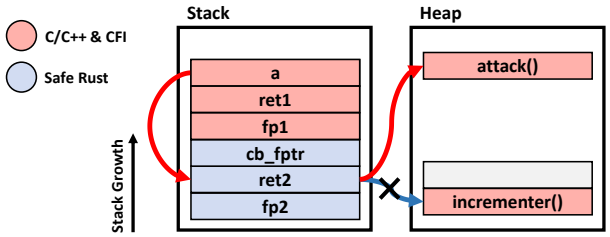


Fig. 5: Since Rust does not deploy a Shadow Stack due to its memory safety, C/C++ can corrupt returns of previously called Rust functions that will never be checked.

B. Rust Bounds Check Bypass

We first demonstrate a variant of CLA that can bypass the simple bounds checks that Rust inserts on certain memory accesses. As mentioned previously, for statically-sized objects in memory, such as arrays, Rust will perform bounds checks on associated memory accesses. In Figure 6a, the `Data` struct contains a field `vals` that is a statically sized array. If Rust

```rust
fn rust_fn(cb_fptr: fn(&mut i64)) {
    // Initialize some data
    let mut x = Data {
       vals: [1,2,3],
       cb: cb_fptr,
    };

    unsafe{ vuln_fn(/*Ptr to x.vals*/) }

    // Uses corrupted function pointer
    (x.cb)(&mut x.vals[0]);
}
```

(a) Rust code that calls C/C++ to modify a Rust struct.

```c
// This function modifies a given array
// Can cause an OOB vulnerability
void vuln_fn(int64_t array_ptr_addr) {
    // These values are set by a corruptible
    // source, e.g., user input
    int64_t array_index = 3;
    int64_t array_value = get_attack();

    int64_t* a = (void *)array_ptr_addr;
    a[array_index] = array_value;
}
```

(b) C/C++ code that performs an Out-of-Bounds (OOB) error.

Fig. 6: Sample code to illustrate how CLA can circumvent the Rust type system to cause a OOB error.

```rust
fn rust_fn(cb_fptr: fn(&mut i64)) {
    let heap_obj: /* Rust heap allocation */

    unsafe{ vuln_fn(/*Ptr to heap_obj*/) }

    heap_obj[0] += 5; // UaF
}
```

(a) Rust code that uses a pointer wrongfully freed by C/C++.

```c
// Frees object it does not own
void vuln_fn(int64_t obj_ptr_addr) {
    int64_t* a = (void *)obj_ptr_addr;

    //C/C++ frees Rust allocated object!
    free(a);
}
```

(b) C/C++ code that leads to a Use-after-Free (UaF) error in Rust.

Fig. 7: Sample code to illustrate how CLA can coerce Rust into causing a UaF error.

```rust
fn rust_fn(cb_fptr: fn(&mut i64)) {
    let fptr: /* Function pointer */

    //C++ code overwrites fptr
    unsafe{ vuln_fn() }

    // No CFI checks!
    fptr();
}
```

(a) Rust code that uses a function pointer.

```c
void vuln_fn() {
    int64_t a[1] = {0}; // C/C++ array
    // These values are set by a corruptible
    // source, e.g., user input
    int64_t array_index = 47;
    int64_t array_value = get_attack();

    // Arbitrary Write to Rust fptr
    a[array_index] = array_value;
}
```

(b) C/C++ that overwrites a Rust function pointer.

Fig. 8: Sample code to show how CLA can corrupt a Rust function pointer to execute a weird machine and circumvent CFI.

were to attempt to access the fourth element of `x.vals`, say on line 13, the program would either completely fail to compile or panic at runtime depending on the optimizations of the Rust compiler. However, when Rust calls `vuln_fn` on line 8, the unsafe C/C++ function is free to access (and modify) the fourth element of `x.vals`. Because the "fourth" element of `x.vals` is actually the function pointer, `x.cb`, in memory, C/C++ is able modify the Rust function pointer, achieving a control-flow hijack and executing a weird machine when Rust later uses the function pointer at line 11. Therefore, when Rust interfaces with FFI, the typical *spatial* memory safety guarantees of Rust may silently fail.

C. Rust Lifetime Bypass

We next demonstrate how CLA can bypass Rusts temporal memory safety guarantees in Figure 7. Rather than relying on the programmer to properly allocate and free memory, the Rust type system attaches a lifetime to each object and frees the object when it goes out of scope. By default, Rust uses the libc `malloc()` implementation, meaning that Rust-C/C++ applications in practice share a heap managed by the same allocator. Thus, C/C++ may deallocate memory without Rust's knowledge. On line 2 in Figure 7a, Rust allocates a heap object.[5] When C/C++ frees this object on line 6 in Figure 7b, Rust still believes this object is alive, and valid for

[5]This can happen automatically, or be forced via the Box<> data structure.

TABLE II: CLA Variants using Multi-Language-Specific Vulnerabilities

New Attack	Memory Corruption Used		Weird Machine Execution Origin	
	Spatial	Temporal	Forward Edge Corruption	Backward Edge Corruption
Corrupt Dynamic Bound	✔		✔	✔
Double Free		✔	✔	
Intended FFI Interactions	✔	✔	✔	✔
Concurrency Safety	✔	✔	✔	✔

use. Consequently, Rust has no problem with the object being used at line 6 of Figure 7a, leading to a UaF vulnerability despite Rust's temporal memory safety guarantees. CLA can thus cause Rust to silently perform a UaF.

D. C/C++ Hardening Bypasses

While we have demonstrated that CLA can bypass the memory safety of Rust, we now show that CLA can also circumvent hardening techniques applied to C/C++ code. In particular, in Figure 8, we illustrate how C/C++ can corrupt a Rust function pointer on the stack which will lead to an opportunity for C/C++ to bypass CFI checks, which are *not* present in Rust code. On line 9 in Figure 8b, C/C++ performs a typical OOB error and corrupts the `fptr` on the Rust stack to point to an attacker chose location. When Rust uses this function pointer on line 8 of Figure 8a, the attacker has successfully hijacked the application's control flow to an arbitrary location to execute a weird machine. If the corrupted pointer had instead been used in C/C++, a CFI check would have detected the deviation from the application's CFG. The memory effects of this bypass are illustrated in more detail in Figure 4, and we can note that Figure 5 demonstrates a similar attack, but one that bypasses the Shadow Stack instead of CFI by overwriting a Rust return value.

While previous work has introduced similar C/C++ hardening bypasses [85], we note that this is only one variant of our presented Cross-Language Attacks (CLA). Most importantly, our work demonstrates that not only can we bypass C/C++ hardening with CLA, but we illustrate how CLA causes Rust memory safety guarantees to be violated. In fact, our goal is to demonstrate that the philosophy of incrementally hardening memory unsafe code with memory safe code can have serious flaws—beyond C/C++ hardening bypasses—if not handled properly.

V. CLA using Multi-Language-Specific Vulnerabilities

Beyond reviving the threat of memory safety vulnerabilities in "safe" languages, and bypassing existing partial memory safety defenses in unsafe languages, multi-language applications are vulnerable to variants of CLA that only arise in the context of multi-language applications. In particular, we highlight four such new vulnerabilities. First, Rust's spatial memory safety can rely on bounds stored in memory which is only safe if the *entire* application is memory safe. Second, Rust's automatic memory management relies on it being the only entity controlling the allocation status of memory. However, Rust commonly uses the libc `malloc()` implementation under the hood, giving rise to vulnerabilities in multi-language applications. Third, we highlight two additional ways intended interactions via FFI over the language barrier can go wrong: passing bad values and more complex serialization/deserialization errors. Finally, we describe how multi-threaded programs heighten vulnerabilities. An overview of the attacks we present in this section is contained in Table II. Moreover, we again point the reader to Appendix A for an illustration of similar examples in Go.

A. Corrupting Rust Dynamic Bounds

For objects whose size is determined at runtime and may change, such as `vectors`, Rust stores the current size of the object in memory. That value is then loaded and used in any required bounds checks. By corrupting the recorded size of the object, an attacker can enable a buffer-overflow of arbitrary length in Rust. While this attack is indirect, we note that Rust is seeing adoption in input processing libraries precisely because of its safety features. Consequently, corrupting the bound of a user facing object may be an efficient way to achieve an arbitrary write in practice. Regardless, this attack primitive is useful for attackers and undercuts Rust's security guarantees in multi-language applications.

Concretely, we demonstrate this attack in Figure 9. Rust allocates a vector on line 3 of Figure 9a. This vector is then passed by reference to the vulnerable C/C++ function, `vuln_fn` in Figure 9b. Because a `vector` in Rust allocates a pointer to the heap for the data in the `vector`, a `capacity` field that denotes the total possible length of the `vector`, and a `len` field that denotes the current length of the `vector` all on the stack at initialization, C/C++ can set the current length of the vector arbitrarily high on line 8 in Figure 9b. Thus, when Rust accesses the 55th element of the `vector` on line 8 of Figure 9a—an obvious OOB access—Rust will not panic as it normally should. Therefore, Rust now operates in the same level of spatial memory safety as C/C++. **Namely, the Rust program—solely written in Safe Rust outside its call to C/C++—can no longer claim spatial memory safety if it successfully compiles.**

B. Double Frees

In Section IV-C we showed how UaF can arise in multi-language application. Here we generalize that to other temporal errors. In particular, if C/C++ frees a Rust object, Rust will still try to free that object at the end of its lifetime, giving rise to a double free vulnerability. Prior work has shown that double frees can lead to exploits in practice [37].

Looking back to our example in Figure 7, even if Rust did not directly use the `heap_obj` on line 6 after calling the `vuln_fn` C/C++ function, when the scope of `rust_fn` finishes, Rust will cleanup any memory associated with `heap_obj`. While Rust lifetimes can become more complex than default with explicit lifetimes and custom Drop traits that

```rust
fn rust_fn(cb_fptr: fn(&mut i64)) {
    //Rust vectors have dynamic bounds
    let mut vecs: vec![4];

    unsafe{ vuln_fn(/*Ptr to vecs*/) }

    // C++ changed vecs size to 128!
    let vec_fp_addr: i64 = x.vecs[55];
}
```

(a) Rust code that passes a vector to C/C++.

```cpp
void vuln_fn(int64_t vec_ptr_addr) {
    // These values are set by a corruptible
    // source, e.g., user input
    int64_t array_index = 2;
    int64_t array_value = 128;

    int64_t* a = (void *)vec_ptr_addr;
    a[array_index] = array_value;
}
```

(b) C/C++ code with an arbitrary write vulnerability.

Fig. 9: Example of C/C++ using an arbitrary write to corrupt the size of Rust vector.

```rust
// Uses a function pointer provided by C/C++
fn rust_fn(cb_fptr: fn(&mut i64)) {
    unsafe { let mut fptr = vuln_cb_fptr(); }
    fptr();
}
```

(a) Rust code that calls C/C++ to receive a callback pointer.

```cpp
// Returns a call back function to register
int64_t vuln_cb_fptr() {
    int64_t fptr = get_attack();
    return fptr;
}
```

(b) C/C++ code that corrupts a return value to Rust.

Fig. 10: Sample code to illustrate how CLA can corrupt even data intended to cross the FFI boundary.

define the cleanup behavior, it will inevitably lead to memory being freed twice. Thus, Rust could then overwrite the memory it just freed, leading to a series of propagating UaF errors, and inevitably, undefined behavior. Therefore, Rust now operates in the same level of temporal memory safety as C/C++. **Namely, the Rust program—solely written in Safe Rust outside its call to C/C++—can no longer claim temporal memory safety if it successfully compiles.**

C. Intended Interactions over FFI

Interactions over FFI, where Rust-C/C++ *intend* to share data can also give rise to CLA. While we saw that the attacks in Section IV are simplified when Rust shares a pointer with C/C++, we observe more complex attacks that can occur when C/C++ shares data with Rust. One version of this is where C/C++ hands Rust a pointer to a buffer to populate (*e.g.,* for user input sanitized by Rust). Another scenario is when Rust receives a function pointer from C/C++ (*e.g.,* for a callback function triggered on some event). In either case, Rust has no way of verifying that the shared pointer—or its contents—is valid, and must trust C/C++. This trust can be abused, leading to CLA.

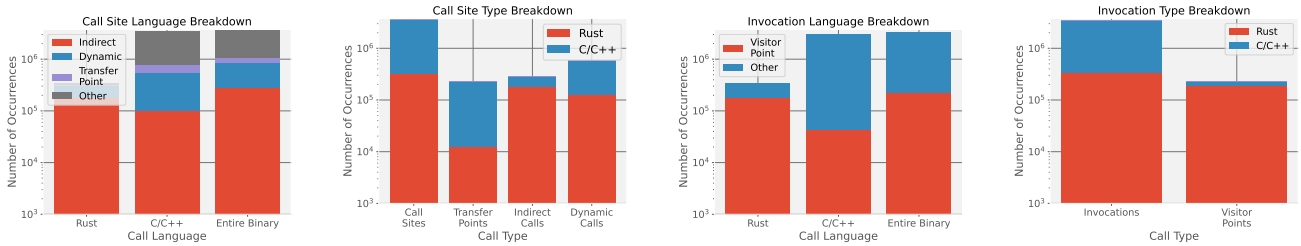
For example, in Figure 10a, Rust calls `vuln_cb_fptr` to ask C/C++ to return a function pointer for its new callback function. If C/C++ returns malicious information, as we see on line 3 in Figure 10b, then when Rust uses that function pointer at line 4 of Figure 10a, the attacker successfully hijacks the application's control and can execute their weird machine. Note that any unsafe function could potentially be used to corrupt the return value from `vuln_cb_fptr`, perhaps on a different thread, or a callee of that function. Rather than pass corrupted data Rust as a return value, another variant of this attack is to have C/C++ directly pass corrupted data to Rust as a function parameter when it invokes a Rust function.

However, the Rust compiler does emit warnings about function pointers crossing the FFI boundary. In particular, Figure 10a is simplified; the programmer must explicitly transmute the function pointer received from C/C++ to a function pointer type within an unsafe block (this requirement is unique to function pointers, as other data can often be coerced with the `as` keyword in Safe Rust). While this helps identify which regions the programmer should likely sanitize, the function pointer conversion can reside within the same unsafe block used to call the C/C++ function, which may lead to bugs being overlooked.

More complex intended interaction errors are of course possible over the FFI interface. For instance, C/C++ strings and Rust strings have different representations, forcing conversion through null terminated C/C++ strings for compatibility over FFI. This illustrates the need for *serialization* over the FFI interface. Serialization is a well known source of errors [87], but it has primarily been considered in inter-application scenarios (*e.g.,* I/O to networks, Files, or IPC), not intra-application scenarios where it arises as a type of CLA.

D. Concurrency and CLA

The preceding examples are all single threaded. This imposes constraints on CLA attacks; typically Rust must call into C/C++. However, in real applications such constraints are less relevant. All threads have access to the entire memory space, meaning that a C/C++ function executing on one thread that contains an arbitrary write can attack a Rust function operating on a separate thread. This effectively removes ordering constraints. **CLA is thus more general than just FFI issues when the multi-language application is multi-threaded.**



(a) Call sites by language. (b) Call sites by type. (c) Invocations by language. (d) Invocations by type.

Fig. 11: Stacked bar plots to breakdown different types of statistics found in Table III. Y-axis is logarithmic.

TABLE III: The prevalence of CLA building blocks. The number is bold is the total number of each item in the binary. (X, Y) represents the breakdown of the counts where X is the fraction of the total items the specific item accounts for, and Y is the fraction of those that come from the specific language. For example, there are 12,118 transfer points in Rust that constitute 3.70% of all call sites in Rust, and 5.32% of the transfer points in the entire binary come from Rust.

(a) Total function metrics.

	Rust	C/C++	Entire Binary
Total Functions	**487,763** (100%, 26.68%)	**1,340,347** (100%, 73.32%)	**1,828,110** (100%, 100%)

(b) Call site metrics

	Rust	C/C++	Entire Binary
Call Sites	**327,653** (100%, 9.23%)	**3,220,415** (100%, 90.77%)	**3,548,068** (100%, 100%)
Transfer Points	**12,118** (3.70%, 5.32%)	**215,778** (6.70%, 94.68%)	**227,896** (6.42%, 100%)
Indirect Calls	**179,598** (54.81%, 64.04%)	**100,843** (3.13%, 35.96%)	**280,441** (7.90%, 100%)
Dynamic Calls	**126,710** (38.67%, 22.15%)	**445,418** (13.83%, 77.85%)	**572,128** (16.13%, 100%)

(c) Invocation metrics

	Rust	C/C++	Entire Binary
Invocations	**346,469** (100%, 10.25%)	**3,032,583** (100%, 89.75%)	**3,379,052** (100%, 100%)
Visitor Points	**184,799** (53.34%, 81.09%)	**43,097** (1.42%, 18.91%)	**227,896** (6.74%, 100%)

VI. Evaluation

In order to demonstrate the applicability of CLA, we collect a series of metrics that demonstrate the prevalence of CLA building blocks in real-world open source code bases. We focus on Mozilla Firefox [47], a large, open source project that has been consistently ported piece-by-piece from C/C++ to Rust. As Mozilla contributes to the creation of both Firefox and Rust itself, we believe characteristics of Firefox will act as a representative showcase for features of Rust and C/C++. We perform static analysis determine the order of magnitude prevalence of CLA building blocks to assess the scope of the problem; our goal is not to build proof-of-concept exploits. Thus, our two research questions for the evaluation are:

RQ1 How prevalent are language transitions?
RQ2 What is the distribution of language transitions across functions (*i.e.,* are language transitions widespread among all functions or centralized in a few)?

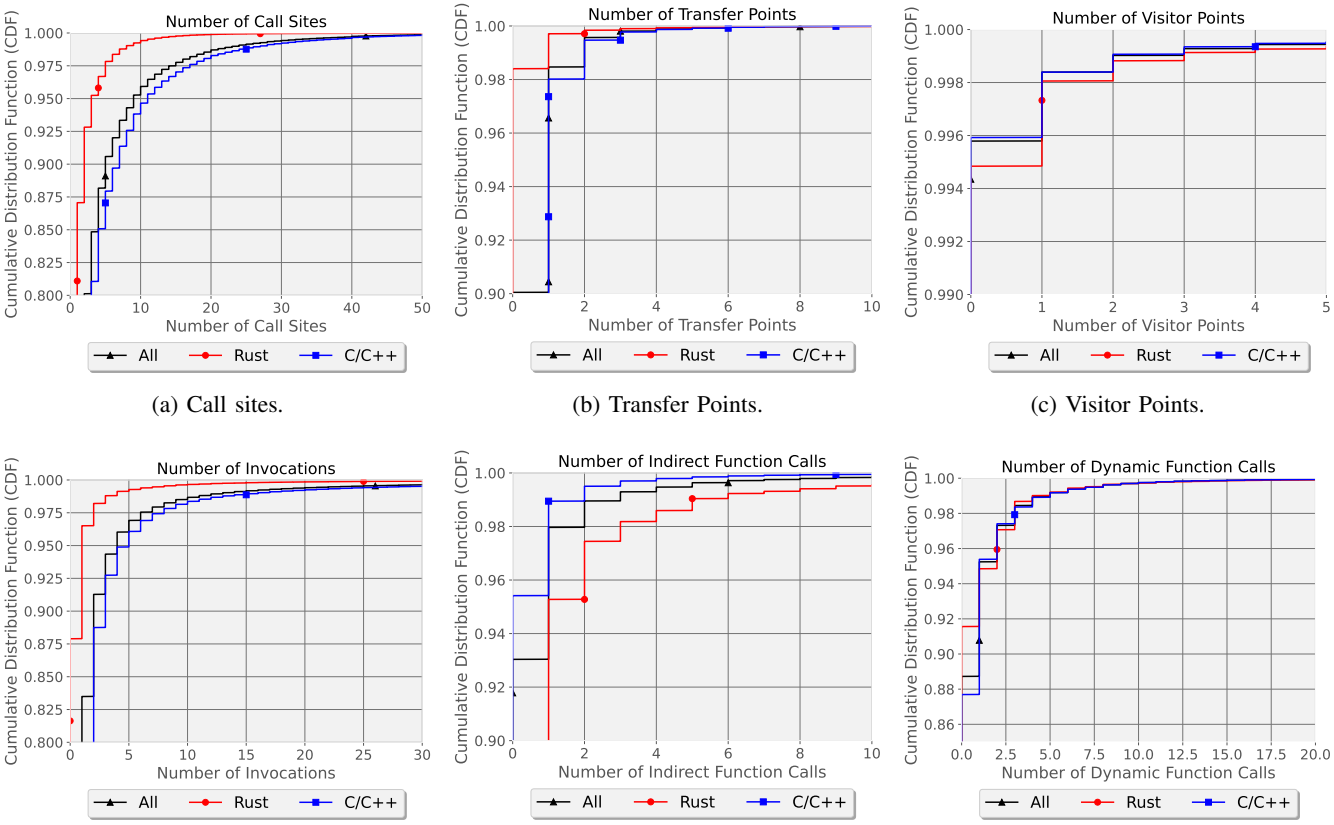
A. Methodology

We analyze Firefox version 92.0a1 using a debug build with optimizations disabled and the Rust v0 name mangling scheme, which allows us to distinguish Rust and C++ mangled function names. Our analysis utilizes pyelftools [21] to parse debug information, and objdump [109] to find callsites. We opt to analyze the compiled file as we can extract all needed information for our evaluation at the binary level and it is simpler than source level analysis. Additionally, we make our analysis openly available online[6].

a) Source Language: We use a novel set of fingerprinting techniques to determine the source language of each function. Our fingerprinting technique is based on the differences in name mangling between languages during the compilation process. Name mangling is used by Rust and C++ to support function overloading, while still presenting unique function names to the linker. Name mangling encodes metadata about the function (*e.g.,* return value and argument types into the function name). For Rust v0 mangling, included with the nightly compiler, the function type (*i.e.,* normal, closure, or monomorphized) and the types of the function signature are encoded in the function name. The standard Rust mangling scheme is the same as C++, so we use the v0 scheme to differentiate the source languages. Note that FFI between Rust and C++ uses C style unmangled function names as the call target. Consequently, we assume that unmangled calls in a known Rust or C++ function represent language transitions (which will include a transition from C++ to C as a language transition). We do not assess the accuracy of this technique; we aim only to obtain a rough order of magnitude understanding of the prevalence of CLA building blocks.

b) Metrics: As discussed in Section III, CLA leverages control flow transitions between source languages which motivates RQ1, and thus, the metrics we collect must quantify the behavior and interactions of functions with other languages. Namely, we want to observe how frequently, to which degree, and in what manner Rust and C/C++ invoke each other.

[6]https://github.com/mit-ll/Cross-Language-Attacks



(a) Call sites. (b) Transfer Points. (c) Visitor Points.

(d) Invocation Points. (e) Indirect Calls. (f) Dynamic Calls.

Fig. 12: The Cumulative Distribution Function (CDF) of each CLA building block metric.

We collect a total of seven metrics to quantify language transitions in Firefox. First, we observe the *total number* of functions in each language to get a sense of how significant the role of new languages such as Rust are in development. Second, we collect the set of functions that each function calls, which we denote as *call targets*, as well as the set of *call sites* (*i.e.,* where control flow changes) within the function. An important subset of call sites are the *transfer points* in which one language is calling a function in a different language. Transfer points are key building blocks of CLA. For each call site, we determine the type of call used. Calls can be: *indirect* (*i.e.,* the call target is in a register), *dynamic* (*i.e.,* indirect through the program lookup table (PLT) used for functions in dynamically linked libraries), or *direct* (*i.e.,* to a constant address). Note that metrics on direct calls are not reported. Indirect calls are frequently targeted by code reuse attacks in the Weird Machine Execution phase, *c.f.,* Figure 3. The PLT can similarly be corrupted, leading to the same effect [124]. For each function, we also collect its *invocation points*, or the set of functions that call it. We are particularly interested in invocation points that come from a different language, which we denote as *visitor points*.

B. Results

First, we obverse the raw totals of our metrics collected in Firefox, including constituent libraries built from the Mozilla repo, in Table III. Table III is made of three subtables that each breakdown related classes of statistics. In each cell of Table III, we show the total number of each metric collected in bold, the percentage of its class of metrics (*i.e.,* transfer points are a subclass of call sites) on the left, and the percentage of the metric that comes from each language with respect to the entire binary on the right. For example, we found 12,118 transfer points in Rust functions with which we can conclude that 3.70% of call sites in Rust are transfer points, but only 5.32% of transfer points come from Rust. These results make sense when paired with the fact that only 9.23% of call sites in Firefox are written in Rust, but have powerful implications for CLA in that nearly one in 25 of all Rust function calls will call back to an unsafe language. In fact, these results demonstrate significant opportunity for CLA. While a function call within a Rust function is typically the target of the memory corruption found in C/C++ for a CLA, in order for the unsafe language to corrupt Rust data, the Rust data must remain "live" while the unsafe language operates. Namely, when Rust calls C/C++, C/C++ has an opportunity to corrupt Rust data and create exploit points. These results indicate this "liveliness" requirement will often be met. Multi-threaded programs, such as Firefox, also relax this requirement as a C/C++ function can corrupt a Rust function on another thread. Moreover, we present a visualization of these statistics in a series of stacked bar plots found in Figure 11. Note that the y-axis in these figures is logarithmic.

We highlight more results from Table III notable to CLA

TABLE IV: Heavy hitter functions in Firefox.

	Rust	C/C++
Top Functions with Call Sites	1. assert_initial_values_match@libxul (**588**) 2. get_longhand_property_value<alloc>@libxul (**464**) 3. get_longhand_property_value<nsstring>@libxul (**459**)	1. CreateInstance@libxul (**1,631**) 2. generateBodyEv@libxul (**1,160**) 3. run@libxul (**846**)
Top Functions with Transfer Points	1. main@crashreporter (**55**) 2. main@modutil (**25**) 3. main@logalloc-replay (**24**)	1. Unified_cpp_protocol_http3@libxul (**84**) 2. UIShowCrashUI@crashreporter (**54**) 3. nsWindow@libxul (**49**)
Top Functions with Invocations	1. as_bytes@libxul.so (**930**) 2. state@libxul (**554**) 3. _Unwind_Resume@plt (**520**)	1. AnnotateMozCrashReason@libxul (**134,254**) 2. ReportAssertionFailure@libxul (**131,545**) 3. Array_RelocateUsingMemutil@libxul (**17,475**)
Top Functions with Visitor Points	1. _Unwind_Resume@std (**488**) 2. as_str_unchecked@libxul (**25**) 3. qcms_transform_data@libxul (**24**)	1. __assert_fail@GLIBC (**4388**) 2. ostream@GLIBC (**3326**) 3. strlen@GLIBC (**1294**)

as follows. We observe that 54.81% of call sites in Rust are indirect calls which indicate a use of a function pointer to make the call. This is important because these function pointers are the target of corruption in the forward-edge version of the CLA. Similarly, 38.67% of call sites in Rust are dynamic, which unsafe languages can corrupt for more serious loss of memory safety in Rust, similar to the attack presented in Figure 3. On the other hand, we see that an overwhelming number of invocations of Rust functions come from memory unsafe languages. Specifically, 53.34% of invocations of Rust functions are visitor points, which indicates that C/C++ functions call Rust functions just as often as other Rust functions. Lastly, we point out that while the percent of these calls are convincing, the magnitude of transfer points (12,118) and visitor points (184,799) in Rust are also significant.

Next, we investigate Figure 12 to illuminate the trends of our metrics over the entire set of functions to compliment the aggregate values of our metrics found in Table III. For example, from Figure 12a and Figure 12d, we see that the degree of unique functions called by Rust and towards Rust respectively overall tends to be less than C/C++ functions, while in Figure 12b and Figure 12c, we see that the subclass of functions we care about (*i.e.,* transfer and visitor points), tends to be similar between Rust and C/C++. Moreover, in Figure 12c, less than 1% of C/C++ functions contain a visitor point. Thus, we can conclude that while frequent, the number of unique functions that Rust calls that cross the language boundary is limited. However, it is important to remember that there is a large number of unique C/C++ functions (1,340,347), so even 1% of these functions being relevant to CLA is still significant.

Because we observe a small number of influential functions, which we denote as *heavy hitters*, with respect to CLA, we next present the top heavy hitters for call sites, transfer points, invocation points, and visitor points in Table IV. In this table, we present the top three heavy hitters for each language with its respective count. The function name is followed by an '@' symbol to indicate in which binary the function resides. Of note, we can see that the top visitor points in C/C++ functions come from `GLIBC`, and have a large magnitude (*e.g.,* 4,388 for __assert_fail). This large magnitude indicates a long tail on the corresponding CDF in Figure 12c and suggests many other functions in C/C++ will have only one or two visitor points. Further, the library *libxul* tends to dominate the heavy hitters list which indicates an ample opportunity for CLA.

Our findings indicate that the building blocks for CLA are abundantly available and an attacker has a very large catalog of options when building such attacks. These results highlight the prevalence of opportunities for CLA and the fact that existing countermeasures are not sufficient to prevent them. New countermeasures are necessary when applications are written in multiple languages with mismatching threat models, which we further discuss in the next section.

VII. Discussion

We have focused our discussion of CLA primarily on Rust and Go in combination with C/C++. However, the issue of threat model mismatches extends beyond these languages, and beyond issues of memory safety in multi-language applications. In this section, we discuss future research directions and the broader implications of CLA. We also provide some thoughts on securing multi-language applications.

A. CLA in Go

In contrast to Rust, which was designed as a systems programming language that would have to integrate with C/C++ to be adopted, Go is intended to largely be a stand alone programming language. While Go supports multi-language applications via CGo, and there are certainly projects that leverage CGo (*e.g.,* Docker, Kubernetes, and CockroachDB), several Go language developers discourage use of CGo have written publicly about issues created by CGo [29] for Go applications. In fact, some projects that initially leveraged CGo, later chose to remove it (*e.g.,* the Go implementation of git [1]). What direction Go will go in the future remains to be seen.

Regardless, the vulnerabilities in Section IV and Section V nearly all directly translate to Go from Rust (*c.f.,* Appendix A). The exception are the temporal safety vulnerabilities. Go leverages garbage collection (GC) to provide temporal safety, in contrast to Rust's lifetimes. This opens the door to use-after-gc errors and more complicated double free scenarios when Go interacts with C memory management.

B. CLA In Other Languages

CLA arises anytime two or more languages are used in an application and have different threat models. This extends beyond just Rust-C/C++ or Go-C/C++. For instance, the C++ language standards since C++11 have introduced a growing

eco-system for safe(r) programming practices, such as smart pointers. However, applications written in C++11 and newer are still backwards compatible with older C++ standards and C applications, and often feature code with older code standards (*e.g.,* in included libraries). Such older, and unsafer, components of the application can subvert the additional safety guarantees offered by newer C++ features. Another instance of this in language dichotomy is applications in Rust that contain unsafe Rust, which has seen recent research interest [18], [69].

Note that while the FFI interface requires the `unsafe` keyword in Rust, CLA is fundamentally much more dangerous than normal uses of unsafe. CLA opens the door to the entire gamut of vulnerabilities present in unsafe languages, whereas unsafe is usually used for relatively simple operations that can't be statically proven safe by Rust's type system, and so are excluded. As Rustbelt [58] has shown, such uses of unsafe are simple enough to be amenable to formal verification, as opposed to the exponentially larger amount of code exposed via FFI than can be leveraged in CLA attacks.

a) Interpreted Languages: Interpreted languages add another dimension to possible threat model mis-matches. The Java Virtual Machine (JVM) has been implemented in both C and C++, and the original Python interpreter is written in C, though others exist today. Indeed – attacks against Java have targeted the JVM [49]. All browsers have JavaScript interpreters, though attempts are made to sandbox them [13], [35]. We leave a deep inspection of appropriate threat models for mixed interpreted/compiled applications, particularly when the interpreted language has safety guarantees that do not align with its interpreter, as future work.

b) Multiple Safe Languages: CLA is possible in multi-language application even if all component languages are themselves safe. For instance, Rust and Go both provide memory safety, but because their *strategy* to provide memory safety differs, an attacker could launch a CLA on such a multi-language program. In particular, Rust and Go prevent temporal memory corruption with lifetimes and garbage collection respectively. Should these different systems disagree on the state of memory, double frees or UaFs are possible. We believe other subtle vulnerabilities are likely, but do not explore them here.

c) CLA and Verified Code: Formal methods increasing maturity is evidenced by the seL4 [64] microkernel and recent verification of KVM [67]. Formal verification offers mathematical proof of certain properties, as long as the assumptions used in the analysis hold and the underlying model is accurate. The current approach is to formally verify a critical subset of code, such as the OS/hypervisor or cryptographic libraries [134]. Such verified code is then used as part of a larger, unverified application, enabling CLA. Note that the mixture of verified and unverified code opens up new attack possibilities not explored here, and the unverified code can be used to undermine any of the assumptions/modeling used during formal verification.

C. CLA Beyond Memory Safety

CLA results from the mixed assumptions of the different threat models in a multi-language application, and this extends beyond memory safety. Different assumptions and threat models around type safety enable type confusion [50], [113] attacks in multi-language applications. This endangers schemes for other purposes, including language-based isolation [23], [70], [78], general verification [62], and information-flow control [75]. Correctness violations are also possible—Rust's type system notably provides significant concurrency safety guarantees. Indeed, a selling point of Rust is "fearless concurrency" [63] in which the type system removes many hard-to-debug concurrency errors. In multi-language applications, CLA can reintroduce data races as Rust's guarantees no longer hold. While more benign than memory safety violations, such errors can still lead to denial-of-service attacks.

Our insight that differing assumptions leads to adverse effects holds generally—multi-language applications must be explicit about the assumptions each component is making, and care must be taken to ensure that the application as a whole is not weaker than its constituent parts.

D. Defense Strategies for CLA

Next, we offer some thoughts on general approaches for defending against CLA, and break the problem into two components: preventing unintended interactions and securing intended interactions. We end with a discussion on alternative approaches.

a) Preventing Unintended Interactions: Preventing unintended interactions requires isolating—to the greatest extent possible—the memory of the different language components. One such approach could place each language component in its own process and rely on virtualization and explicit shared memory to control interactions. In fact, this has been proposed by Sandcrust [65]. Doing so is likely impractical, however, particularly if more than two languages are involved. Consequently, intra-process isolation techniques will need to be refined for this use case. For example, recent work shows that isolating the heap between languages is a significant step in the right direction [91]. However, doing so will not be free, as there will be performance costs both for the isolation technique and to allow data-flow across the language boundary. Isolating the stack may require separate stacks per language, which will increase the overhead of language transitions and requiring more intrusive changes. Finally, while code isolation already exists outside of JIT systems courtesy of DEP [15], preventing the corruption of generated code has also been studied for JIT [127], and is largely a separate problem.

b) Securing Intended Interactions: Securing intended interactions requires verifying that interactions across the FFI interface cannot violate the security properties of either language. To date, there has been work on using formal methods [58], [89], [122] to do so in a variety of contexts. In particular, Rust FFI has been studied, building on existing literature (*e.g.,* the java native interface (JNI) [115]). Beyond formal methods, new sanitizers that target interactions across FFI should be developed by the community, as well as runtime defenses. For example, in order to sanitize data meant to be shared between Rust and C/C++, Rivera et al. propose a new pointer construct called *pseudo-pointers* [91]. Moreover, tagged architectures [43], [92], [107], [129] that enable metadata to be added to data provide a promising hardware based approach for mitigating CLA as well. Indeed, the Cheri project [130] has proposed such an extension [126].

c) Alternative Defenses: Some recent work attempts to solve specific subsets of CLA. For example, there is an ongoing effort to add CFI checks to Rust code (*e.g.,* Control Flow Guard (CFG) [16]). Such a defense will help mitigate specific variants of CLA, such as the attack outlined in Section IV-D, since defense assumptions between the two languages will now match for this particular variant. However, as we have mentioned, CLA proposes a problem much deeper than control-flow hijacking (*e.g.,* loss of Rust memory safety guarantees) that such a defense cannot solve. Similarly, incremental deployment of randomization techniques, such as Address Space Layout Randomization (ASLR) [22], [86], offers another alternative approach, but previous work has shown randomization can be bypassed [95], [102]. Moreover, it is unclear if ASLR can address each variant of CLA detailed in Section IV and Section V. Thus, we believe that while these defenses have merit, they address a symptom of CLA rather than the cause.

VIII. Related Work

In this section, we survey a broad range of related work that touches on issues related to either CLA attacks or potential mitigations thereof.

a) Formal Methods and Rust: Work on using formal methods to secure Rust can be broken into two threads: work on securing unsafe Rust [32], [58]–[60], and work on securing FFI interfaces to Rust in secure enclaves [121]–[123]. Rustbelt [58] provided the first formal model of Rust, and verified a simplified version of the language, showing that Rust's claimed security properties hold. Further, Rustbelt has verified many data structures provided by the Rust standard library, showing that the provided abstractions are indeed safe, even though the underlying implementation uses unsafe Rust. In contrast to such work, CLA considers the security of unverified code, which will remain the norm for large applications. Work on verifying FFI interfaces is limited to the enclave context, where the interfaces are simpler, and intended interactions through FFI. CLA shows the perils of unintended interactions.

b) Isolation Solutions: There is a large body of work on providing intra-process isolation, either in software [39], [44], [45], [65], [76], [100], [132] or in hardware [33], [34], [36], [93], [111], [125], [126], [130]. On the software side, Sandcrust [65] separates C and Rust into separate processes and relies on RPC for communication. As described in Section VII this is a heavy weight and impractical solution. Prior work on hardware security provides variously flexible security policies, ranging from bounds checks [34] to unlimited tags with arbitrary policies [36]. The Cheri project [130] is the most successful of these, and has demonstrated that Cheri can isolate Rust [126] in multi-language applications.

c) Enclaves: Enclaves have seen significant recent interest as a way to remove the OS from the TCB. Intel's SGX [17], [38], [61], [68], [74], [83], [98], [104], [105], [118] and ARM's TrustZone [19], [24], [28], [55], [88], [96], [112], [116], [120], [133] have both seen considerable research on both their defensive applications, and flaws in their security guarantees, and grew out of prior academic work [31], [107]. Enclaves have also been adopted for embedded contexts [128]. The enclaving programming model is too restrictive for large applications such as browsers, though it could in theory help address CLA if developers are willing to enclave the safe language component.

d) Language Safety: Programming language safety has seen extensive study, including areas such as the Java Native Interface [30], [51], [106], [115], safe subsets of C [56], [79], and security solutions building on Rusts guarantees [18], [20], [78], [131]. All of these works look at security guarantees of programming languages, including systems that can be built on top of them [78], or properties that underlie those guarantees [131]. Many focus on the security of individual languages [18], [56], [79], or attacks thereon [51]. Papaevripides et al. [85] build an end-to-end CLA exploit on Firefox, an instance of the attack in Section IV-D. In contrast, we are the first work that broadly considers the security implications of multi-language applications and improperly mixed language threat models.

IX. Conclusion

In this paper, we present the first security model for applications developed in multiple programming languages. We specifically focus on cases where the application is written well (limited or no usage of `unsafe` code in the parts written in the safe language) and where advanced protections are applied to the parts written in the unsafe language. We illustrate that because of mismatching threat models, an attacker can maneuver between various stages of an exploit in such a way that avoids triggering safety/hardening checks, while succeeding in hijacking control. Dubbed Cross-Language Attacks (CLA), these attacks can non-intuitively result in weakening of the overall application security when parts of the application are ported to a safe programming language. We illustrate different variants of CLA and performed automated analysis on large code bases to measure the prevalence of CLA building blocks. Our findings illustrate that CLA building blocks are abundantly found in Firefox, and that a new class of countermeasures must be developed to secure multi-language applications, and sketch their design goals for such future defenses.

References

[1] "Go-git," https://github.com/go-git/go-git.

[2] (2015) Why go was the right choice for cockroachdb. https://www.cockroachlabs.com/blog/why-go-was-the-right-choice-for-cockroachdb/.

[3] (2021) Bolt. https://github.com/boltdb/bolt.

[4] (2021) cgo. https://pkg.go.dev/cmd/cgo.

[5] (2021) Docker. https://github.com/docker/.

[6] (2021) Dogear. https://github.com/mozilla/dogear.

[7] (2021) Kubernetes. https://github.com/kubernetes.

[8] (2021) Mesalock linux. https://github.com/mesalock-linux/mesalock-distro.

[9] (2021) mp4parse-rust. https://github.com/mozilla/mp4parse-rust.

[10] (2021) Neqo, an implementation of quic written in rust. https://github.com/mozilla/neqo.

[11] (2021) Redox operating system. https://www.redox-os.org/.

[12] (2021) Servo. https://github.com/servo.

[13] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "Jsand: complete client-side sandboxing of third-party javascript without browser modifications," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 1–10.

[14] Alexander Færøy. (2020) Rustintor. https://gitlab.torproject.org/legacy/trac/-/wikis/RustInTor.

[15] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[16] Andrew Paverd. (2020) Control flow guard for clang/llvm and rust. https://msrc-blog.microsoft.com/2020/08/17/control-flow-guard-for-clang-llvm-and-rust/.

[17] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "Scones secure linux containers with intel sgx," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 689–703.

[18] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[19] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 90–102.

[20] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System programming in rust: Beyond safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 156–161.

[21] E. Bendersky. (2012) Pyelftools. https://github.com/eliben/pyelftools.

[22] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM Computer and Communications Security (CCS'15)*, Oct 2015.

[23] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong, "Theseus: an experiment in operating system structure and state management," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 1–19.

[24] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.

[25] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[26] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 985–999.

[27] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[28] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*, 2020, pp. 18–20.

[29] D. Cheney, "Cgo is not go," https://dave.cheney.net/2016/01/18/cgo-is-not-go.

[30] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son *et al.*, "Cheri jni: Sinking the java security model into the c," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 569–583, 2017.

[31] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 857–874.

[32] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer, "Rustbelt meets relaxed memory," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–29, 2019.

[33] A. A. De Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-policies: Formally verified, tag-based security monitors," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 813–830.

[34] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 103–114, 2008.

[35] A. Dewald, T. Holz, and F. C. Freiling, "Adsandbox: Sandboxing javascript to fight malicious websites," in *proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 1859–1864.

[36] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "Pump: a programmable unit for metadata processing," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, 2014, pp. 1–8.

[37] I. Dobrovitski, "Exploit for cvs double free () for linux pserver," 2003.

[38] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren, "Lightbox: Full-stack protected stateful middlebox at lightning speed," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2351–2367.

[39] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 75–88.

[40] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point (er): On the effectiveness of code pointer integrity," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 781–796.

[41] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 901–913.

[42] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi, "On the Effectiveness of Type-based Control Flow Integrity," in *Proceedings of IEEE Annual Computer Security Applications Conference (ACSAC'18)*, Dec 2018.

[43] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, "HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM Press, 2018, pp. 1583–1600.

[44] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86." in *USENIX Annual Technical Conference*. Boston, MA, 2008, pp. 293–306.

[45] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion, "Enclosure: language-based restriction of untrusted libraries," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 255–267.

[46] R. Gil, H. Okhravi, and H. Shrobe, "There's a hole in the bottom of the c: On the effectiveness of allocation protection," in *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 2018, pp. 102–109.

[47] GitHub. (2021) How much rust in firefox? https://4e6.github.io/firefox-lang-stats/.

[48] Google. (2021) Git repositories on fuchsia. https://fuchsia.googlesource.com/.

[49] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *2003 Symposium on Security and Privacy, 2003.* IEEE, 2003, pp. 154–165.

[50] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. Van Der Kouwe, "Typesan: Practical type confusion detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 517–528.

[51] Y. He, Y. Zhou, Y. Zhou, Q. Li, K. Sun, Y. Gu, and Y. Jiang, "Jni global references are still vulnerable: Attacks and defenses," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[52] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[53] ——, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[54] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the*

2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 1868–1882.

[55] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment." in *NDSS*, 2015.

[56] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: a safe dialect of c." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[57] Joab Jackson. (2020) Microsoft: Rust is the industry's 'best chance' at safe systems programming. hhttps://thenewstack.io/microsoft-rust-is-the-industrys-best-chance-at-safe-systems-programming/.

[58] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–34, 2017.

[59] ——, "Safe systems programming in rust: The promise and the challenge," *Communications of the ACM*, 2020.

[60] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 637–650, 2015.

[61] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[62] O. Kirzner and A. Morrison, "An analysis of speculative type confusion vulnerabilities in the wild," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[63] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[64] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[65] B. Lamowski, C. Weinhold, A. Lackorzynski, and H. Härtig, "Sandcrust: Automatic sandboxing of unsafe components in rust," in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, 2017, pp. 51–57.

[66] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 276–291.

[67] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A secure and formally verified linux kvm hypervisor," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2021.

[68] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for intel {SGX}," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 285–298.

[69] P. Liu, G. Zhao, and J. Huang, "Securing unsafe rust programs with xrust," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 234–245.

[70] S. Maffeis and A. Taly, "Language-based isolation of untrusted javascript," in *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 2009, pp. 77–91.

[71] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[72] J. Meyerson, "The go programming language," *IEEE software*, vol. 31, no. 5, pp. 104–104, 2014.

[73] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," 2019.

[74] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[75] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 228–241.

[76] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[77] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the firefox renderer," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 699–716.

[78] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev, "Redleaf: Isolation and communication in a safe operating system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 21–39.

[79] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[80] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *Security Privacy, IEEE*, vol. 12, no. 2, pp. 16–26, Mar 2014.

[81] H. Okhravi, "A cybersecurity moonshot," *IEEE Security & Privacy*, vol. 19, no. 3, pp. 8–16, 2021.

[82] H. Okhravi, N. Burow, R. Skowyra, B. Ward, S. Jero, R. Khazan, and H. Shrobe, "One giant leap for computer security," *Security Privacy, IEEE*, 2020.

[83] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting sgx enclaves from practical side-channel attacks," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 227–240.

[84] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[85] M. Papaevripides and E. Athanasopoulos, "Exploiting mixed binaries," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 2, pp. 1–29, 2021.

[86] T. PaX, "Pax address space layout randomization (aslr)," *http://pax. grsecurity. net/docs/aslr. txt*, 2003.

[87] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2005, pp. 124–145.

[88] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 195–209.

[89] E. Reed, "Patina: A formalization of the Rust programming language," *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.

[90] Rick Hudson. (2018) Getting to go: The journey of go's garbage collector. https://blog.golang.org/ismmkeynote.

[91] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping safe rust safe with galeed," in *Annual Computer Security Applications Conference*, 2021, pp. 824–836.

[92] N. Roessler and A. DeHon, "Protecting the stack with metadata policies and tagged hardware," in *2018 IEEE Symposium on Security and Privacy*. IEEE, May 2018, pp. 478–495.

[93] ——, "Protecting the stack with metadata policies and tagged hardware," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 478–495.

[94] rsc. (2014) Do not let go pointers end up in c. https://github.com/golang/go/issues/8310.

[95] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz *et al.*, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity." in *NDSS*, 2017.

[96] K. Ryan, "Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 181–194.

[97] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty

of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[98] F. Schwarz and C. Rossow, "Seng, the sgx-enforcing network gateway: Authorizing communication from shielded clients," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 753–770.

[99] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 54–65.

[100] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[101] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.

[102] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 298–307.

[103] R. Shapiro, S. Bratus, and S. W. Smith, "Weird machines," in *In Proceedings of the 7th USENIX Workshop on Offensive Technologies (WOOT)*, 2013, p. 11.

[104] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 955–970.

[105] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves." in *NDSS*, 2017.

[106] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the native beast of the jvm," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 201–211.

[107] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "Hdfi: Hardware-assisted data-flow isolation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 1–17.

[108] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1275–1295.

[109] M. Stevanovic, "Linux toolbox," in *Advanced C and C++ Compiling*. Springer, 2014, pp. 243–276.

[110] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, 2009, pp. 1–8.

[111] G. T. Sullivan, A. DeHon, S. Milburn, E. Boling, M. Ciaffi, J. Rosenberg, and A. Sutherland, "The dover inherently secure processor," in *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*. IEEE, 2017, pp. 1–5.

[112] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 367–378.

[113] J. F. Switzer, "Preventing ipc-facilitated type confusion in rust," Master's thesis, Massachusetts Institute of Technology, 2020.

[114] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[115] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang, "Safe java native interface," in *Proceedings of IEEE International Symposium on Secure Software Engineering*, vol. 97. Citeseer, 2006, p. 106.

[116] A. Tang, S. Sethumadhavan, and S. Stolfo, "Clkscrew: exposing the perils of security-oblivious energy management," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1057–1074.

[117] Tino Caer. (2020) How microsoft is adopting rust. https://medium.com/@tinocaer/how-microsoft-is-adopting-rust-e0f8816566ba.

[118] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 645–658.

[119] P. Wagle, C. Cowan *et al.*, "Stackguard: Simple stack smash protection for gcc," in *Proceedings of the GCC Developers Summit*, 2003, pp. 243–255.

[120] S. Wan, J. Sun, K. Sun, N. Zhang, and Q. Li, "Satin: A secure and trustworthy asynchronous introspection on multi-core arm processors," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 289–301.

[121] S. Wan, M. Sun, K. Sun, N. Zhang, and X. He, "Rustee: Developing memory-safe arm trustzone applications," 2020.

[122] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards memory safe enclave programming with rust-sgx," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2333–2350.

[123] P. Wang, Y. Ding, M. Sun, H. Wang, T. Li, R. Zhou, Z. Chen, and Y. Jing, "Building and maintaining a third-party library supply chain for productive and secure sgx enclave development," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[124] B. C. Ward, R. Skowyra, C. Spensky, J. Martin, and H. Okhravi, "The leakage-resilience dilemma," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 87–106.

[125] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[126] N. Wei and S. Sim, "Strengthening memory safety in rust: exploring cheri capabilities for a safe language," in *Master's Thesis: Wolfson College*, 2020.

[127] T. Wei, T. Wang, L. Duan, and J. Luo, "Secure dynamic code generation against spraying," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 738–740.

[128] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v." in *NDSS*, 2019.

[129] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665740

[130] ——, "The cheri capability model: Revisiting risc in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 457–468.

[131] H. Xi and F. Pfenning, "Eliminating array bound checking through dependent types," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998, pp. 249–257.

[132] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 79–93.

[133] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 72–90.

[134] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

APPENDIX

In this section, we provide code examples of CLA in Go-C/C++ applications that correlate to the code examples presented in Section IV and Section V. In particular, Figure 13, Figure 14, and Figure 15 correspond to the attacks presented in Section IV-B, Section IV-C, Section IV-D respectively while Figure 16, and Figure 17 correspond to the attacks presented in Section V-A and Section V-C respectively.

```go
func go_fn(cb_fptr *func(*int64)) {
    // Initialize some data
    x := Data {
       vals: [3]int64{1,2,3},
       cb: cb_fptr,
    }

    C.vuln_fn(/*Ptr to x.vals*/)

    // Uses corrupted function pointer
    (*x.cb)(&x.vals[0])
}
```

(a) Go code that calls C/C++ to modify a Go struct.

```c
// This function modifies a given array
// Can cause an OOB vulnerability
void vuln_fn(int64_t array_ptr_addr) {
    // These values are set by a corruptible
    // source, e.g., user input
    int64_t array_index = 3;
    int64_t array_value = get_attack();

    int64_t* a = (void *)array_ptr_addr;
    a[array_index] = array_value;
}
```

(b) C/C++ code that performs an Out-of-Bounds (OOB) error.

Fig. 13: Sample code to illustrate how CLA can circumvent Go to cause a OOB error.

```go
func go_fn(cb_fptr *func(*int64)) {
    heap_obj := new(/* Go heap allocation */)

    C.vuln_fn(/*Ptr to heap_obj*/)

    heap_obj[0] += 5 // UaF
}
```

(a) Go code that uses a pointer wrongfully freed by C/C++.

```c
// Frees object it does not own
void vuln_fn(int64_t obj_ptr_addr) {
    int64_t* a = (void *)obj_ptr_addr;

    //C/C++ frees Go allocated object!
    free(a);
}
```

(b) C/C++ code that leads to a Use-after-Free (UaF) error in Go.

Fig. 14: Sample code to illustrate how CLA can coerce Go into causing a UaF error.

```go
func go_fn(cb_fptr *func(*int64)) {
    fptr := /* Function pointer */

    //C++ code overwrites fptr
    C.vuln_fn()

    // No CFI checks!
    (*fptr)()
}
```

(a) Go code that uses a function pointer.

```c
void vuln_fn() {
    int64_t a[1] = {0}; // C/C++ array
    // These values are set by a corruptible
    // source, e.g., user input
    int64_t array_index = 47;
    int64_t array_value = get_attack();

    // Arbitrary Write to Rust fptr
    a[array_index] = array_value;
}
```

(b) C/C++ that overwrites a Go function pointer.

Fig. 15: Sample code to show how CLA can corrupt a Go function pointer to execute a weird machine and circumvent CFI.

```go
func go_fn(cb_fptr *func(*int64)) {
    //Go slices have dynamic bounds
    slice := []int64{4, 5}

    C.vuln_fn(/*Ptr to slice*/)

    // C++ changed the slice size to 128!
    slice_fp_addr := slice[55]
}
```

(a) Go code that passes a slice to C/C++.

```c
void vuln_fn(int64_t slice_ptr_addr) {
    // These values are set by a corruptible
    // source, e.g., user input
    int64_t array_index = 2;
    int64_t array_value = 128;

    int64_t* a = (void *)slice_ptr_addr;
    a[array_index] = array_value;
}
```

(b) C/C++ code with an arbitrary write vulnerability.

Fig. 16: Example of C/C++ using an arbitrary write to corrupt size of a Go slice.

```go
// Uses a function pointer provided by C/C++
func go_fn(cb_fptr *func(*int64)) {
    fptr := C.vuln_cb_fptr()
    (*fptr)()
}
```

(a) Go code that calls C/C++ to receive a callback pointer.

```c
// Returns a call back function to register
int64_t vuln_cb_fptr() {
    int64_t fptr = get_attack();
    return fptr;
}
```

(b) C/C++ code that corrupts a return value to Go.

Fig. 17: Sample code to illustrate how CLA can corrupt even data intended to cross the FFI boundary.