Exploiting Mixed Binaries

MICHALIS PAPAEVRIPIDES and ELIAS ATHANASOPOULOS, University of Cyprus, Cyprus

Unsafe programming systems are still very popular, despite the shortcomings due to several published memory-corruption vulnerabilities. Toward defending memory corruption, compilers have started to employ advanced software hardening such as Control-flow Integrity (CFI) and SafeStack. However, there is a broad interest for realizing compilers that impose memory safety with no heavy runtime support (e.g., garbage collection). Representative examples of this category are Rust and Go, which enforce memory safety primarily statically at compile time.

Software hardening and Rust/Go are promising directions for defending memory corruption, albeit combining the two is questionable. In this article, we consider hardened *mixed* binaries, i.e., machine code that has been produced from different compilers and, in particular, from *hardened* C/C++ and Rust/Go (e.g., Mozilla Firefox, Dropbox, npm, and Docker). Our analysis is focused on Mozilla Firefox, which outsources significant code to Rust and is open source with known public vulnerabilities (with assigned CVE). Furthermore, we extend our analysis in mixed binaries that leverage Go, and we derive similar results.

The attacks explored in this article *do not* exploit Rust or Go binaries that depend on some legacy (vulnerable) C/C++ code. In contrast, we explore how Rust/Go compiled code can stand as a vehicle for bypassing hardening in C/C++ code. In particular, we discuss CFI and SafeStack, which are available in the latest Clang. Our assessment concludes that CFI can be completely nullified through Rust or Go code by constructing much simpler attacks than state-of-the-art CFI bypasses.

CCS Concepts: • **Security and privacy** → **Browser security**; **Software security engineering**;

Additional Key Words and Phrases: Memory safety, Rust, Go, CFI, SafeStack

1 INTRODUCTION

Software can be written in unsafe or safe programming systems. Unsafe systems allow software to utilize freely the address space, and therefore memory corruption, due to wrong programming decisions, is possible. Memory corruption can be combined with several techniques for building exploits that can compromise and fully control a vulnerable program [73]. Despite the rise of safe systems, unsafe systems are still prevalent today [60], since (a) they are faster, (b) a huge (unsafe) code

base is already out there and needs maintenance, and (c) this free-access memory model is attractive, especially for low-level software, such as operating systems, compilers, or even web browsers.

For addressing the security issues of unsafe code (typically C/C++ binaries without runtime memory management), we employ software hardening. Technically, this involves taking an unsafe system and adding safety on top of it by means of introducing certain checks that prevent a vulnerability from becoming an exploit. Several methods have been proposed for software hardening and some of them are now included in most of the available CPUs, operating systems, and compilers. As a short list, all modern CPUs support non-executable data [37], all modern OSes support process relocation and randomization [66], and many modern compilers support stack canaries [46] and some more recent hardening proposals, namely CFI [2] and SafeStack [3].

However, researchers have also considered the possibility of creating fast, system-oriented programming languages, which realize safety with no runtime support. [1] This is certainly attractive, since programs written using these systems are generically fast, inherit safety by design, and if they want to be more flexible in terms of memory access, they can take a risk, which is still *partial*. One such example is Rust [59], a programming language/system developed by Mozilla. Rust realizes *all* safety checks at compile time. This means, that the Rust compiler ensures *statically* that no memory corruption is possible by emitting certain code in the final binary or constraining the use of pointers. The programmer is allowed to use certain directives to escape the constraints imposed by the compiler, but this should be done *explicitly* (e.g., by writing code in an `unsafe` block). In a similar fashion, another example is Go [9], which is a fast systems language with many security protections, primarily developed by Google.

Additionally, code written in Rust or Go can co-exist with (unmanaged) code written in C/C++. This makes these languages further attractive, since several components of an unsafe system can be outsourced to Rust or Go and, thus, become *safe*. Moreover, both Rust and Go are shown to be highly preferred among developers. In particular, in the 2020 edition of an annual survey conducted by Stack Overflow [13], Rust and Go are ranked first and fifth, respectively, for the most loved programming language. As the survey mentions, Rust holds the first place in the aforementioned list for five consecutive years. This shows that Rust and Go are starting to gain wider adoption in the community. As a concrete result, more projects have started replacing *unsafe* code using Rust or Go. For example, Mozilla Firefox, one of the most well-known and established web browsers, is written primarily in C/C++, however, several components are being migrated to Rust *explicitly for safety reasons* [11, 28]. Key modules are the CSS engine (now replaced by Stylo), the rendering module (now replaced by parts of Servo), and the WebAssembly compiler (now realized by Crane).

In this article, we perform a thorough look at the security gains that stem from outsourcing certain unsafe parts of a *hardened* system to a safe system, such as Rust and Go. Typically, one expects that (a) Rust/Go code that calls unsafe code should be considered dangerous and (b) unsafe code that implements parts of the functionality in Rust/Go should be considered safer compared to having all code base written in C/C++. In this article, *we are not interested in (a)*, since it is well established that when Rust/Go calls unsafe code, memory safety is not guaranteed. Instead, we thoroughly deal with (b) and we conclude that there is no trivial answer, while the technical details are important. That is, there are certain cases where the security of a binary is degraded exactly due to the *safe part*, which can be realized in Rust or Go.

---

[1]In this article, runtime support is considered code that runs in parallel with the program and takes certain decisions (e.g., a garbage collector). Code that is produced statically at compile-time and may kick in, when certain conditions are met, at runtime (e.g., an array's bound check), is not considered as runtime support. We explore both options, namely Go, which implements a lightweight garbage collector, and Rust, which implements the notion of *the borrow checker* to eliminate the need for a garbage collector.

For arguing about the security of mixing safe and unsafe code, we consider C/C++ binaries with modern hardening techniques in place. We further investigate the possibilities of using *safe* parts of the code to *hijack*, otherwise secure due to hardening, parts of the *unsafe* code. In other words, we argue that a hardened C/C++ binary is *more secure* than a similar one with outsourced parts in Rust. Precisely, we explore how certain hardening mechanisms, CFI and SafeStack in Clang, behave when Rust/Go code is present in the process' address space. We experimentally show that CFI can be bypassed using Rust/Go code for developing clearly more straightforward exploits than current state-of-the-art CFI bypasses [57, 70]. Our exploits do not rely on respecting the statically computed Control-flow Graph (CFG) for bypassing the enforced CFI and are much *simpler* to implement. However, we demonstrate that SafeStack, in practice, is much harder to bypass using Rust or Go.

Throughout this article, we consider hardened *mixed* binaries. These are binaries that have been produced by compilers of different memory-access models. In particular, we consider mixed binaries of (hardened) C/C++ and Rust or Go. A typical binary of this category is a recent version of Mozilla Firefox. Despite the fact that there is an increasing number of companies (e.g., Dropbox, yelp [12] and npm [26]) that re-write parts of their software in Rust, we chose to focus our analysis on Mozilla Firefox. This stems from the fact that Mozilla Firefox is not only a very well-known *open source* project but also because we have *publicly available vulnerabilities (CVE)*. This is important, because it allowed us to deliver a proof-of-concept exploit based on a *real-world* vulnerability to demonstrate the severity of such attacks. Mozilla is the company that develops both Firefox and Rust, which makes Firefox a good reference project for other projects to look at when trying to combine C/C++ with Rust. As a result, Mozilla Firefox is a good landmark to examine mixed binaries. Furthermore, we extend our analysis in mixed binaries that utilize Go instead of Rust and we derive similar results. In principle, our techniques can work directly to binaries and exploit proprietary software, as well, however reverse engineering often is prohibited by the terms of use of programs. Therefore, for making it easier for the community to reproduce our findings without violating the terms of use of software, we focus on exploiting open-source projects (Mozilla Firefox and go-stats).

In this article, we analyze mixed binaries and we explore if it is possible to bypass hardening using the unhardened *but safe* part. We develop several techniques that are based on fundamental properties of Rust/Go and aim to simply bypass hardening countermeasures imposed in the rest of the mixed binary. Again, we stress here that in the absence of the safe part (Rust/Go code), the unsafe part cannot be trivially exploited.

How fundamental are these attacks? In this article, we explore the feasibility of making an unfortunate combination in security defenses. A C/C++ binary can be hardened using standard compiler features, on the other hand, some code can be re-written in a safe programming language. Both these directions increase the security of the program, but their combination may have the exact opposite result. As we show in this article, re-writing parts of a program in a safe system should be carefully combined with *some* hardening features that the safe system is not aware of, since the safe part can render the hardening less effective. One natural observation is that Rust/Go can be hardened in the same fashion of C/C++. However, this has several shortcomings. First, both Rust and Go realize different programming-language concepts for enforcing memory safety at compile time. In other words, these systems follow a different approach to counter memory errors. Second, adding standard hardening techniques to these systems will introduce additional overheads, which are not, at the moment, explored. Third, applying hardening techniques, such as CFI, for instance, in a *mixed* binary is not currently explored and cannot be trivially enabled. For instance, considering that the Rust compiler targets LLVM, enabling CFI by using the -fsanitize=cfi is not possible, since an entirely different analysis needs to be carried out for resolving the targets

of all indirect branches originating from Rust. Finally, this analysis should be merged with the analysis of the C/C++ code. Therefore, hardening safe systems has to consider several aspects (so far unexplored), such as performance, accuracy, and additional labor to perform the analysis of the CFG. We further discuss this issue in Section 8.2. Finally, we stress that, at least, the Rust community has explored enabling some hardening techniques, but has done so only for the very basic ones [15, 16, 23], and not for any of the advanced ones available today [17, 19, 31].

1.1 Contributions

In this article, we make the following contributions:

(1) We argue that outsourcing certain functionality of an unsafe program to safe code may actually degrade the overall security of the program. Especially for hardened (unsafe) processes, the consequences may be negative, since vulnerabilities of the unsafe part, which otherwise could be contained by the hardening (e.g., CFI/SafeStack) involved, can be *laundered* through the safe part to attack the whole process.

(2) We demonstrate Contribution (1) through a proof-of-concept exploit, based on CVE-2018-6126 [22], which can bypass state-of-the-art hardening, namely CFI (available in all recent versions of Clang) and attack the browser, through the Rust sub-part of Mozilla Firefox. Using a similar approach, we construct proof-of-concept attacks for bypassing SafeStack (also available in all recent versions of Clang), and we conclude that such code patterns are unlikely to be found in actual software. We experimentally show that in Mozilla Firefox. We extend our analysis in Go [9] and we derive similar results by exploiting go-stats [8].

(3) The PoC exploits of Contribution (2) are *not* based on using Rust to call code, which is explicitly annotated as *unsafe* or using Go to call code, which is explicitly imported using import ''C'' by the developer, but on standard Rust/Go code constructs (we explore all of them for *both* systems). In fact, some exploits require *zero* interaction between C/C++ and Rust/Go. This is important, since the part of Rust/Go that is manipulated to deliver the attack is considered explicitly *safe* and it is explicitly used for *increasing* the safety of the program and not for *degrading* it [11, 28].

(4) Finally, we deliver tools that can scan Rust/Go software for identifying *safe* code that, if co-located with (hardened) unsafe code, can be leveraged to exploit vulnerabilities of the unsafe code.

1.2 Paper Organization

Beyond the standard sections, this article is organized as follows. Section 2 can be skipped by anyone with basic understanding of memory corruption in unsafe systems, hardening, and safety imposed at compile time (just make sure to skim the threat model, as presented in Section 2.4). Section 3 is important to understand the fundamentals of the attack presented in this article, while Section 4 and Section 5 can be read only from those interested in diving more in the low-level constructs used by Rust and Go compilers, respectively (a good read of Section 3 can help, and then skimming Section 4 and Section 5 should be just fine). In Section 6, we outline a methodology for finding attack primitives in actual code, such as web browsers, and in Section 7 we show how our findings can be used to compromise a hardened Mozilla Firefox through Rust using an exploit based on CVE-2018-6126 [22]. In Section 8, we present a PoC exploit for a hardened mixed binary attacked through the Go library go-stats, we also discuss several important points that stem from the PoC exploits presented and we discuss the major differences between the two memory models in terms of compile time and runtime checks. In Section 9, we discuss some techniques that could potentially mitigate the attacks discussed in this article. The rest of the article has the typical structure.

2 BACKGROUND AND THREAT MODEL

This section provides some background information in exploiting unsafe systems, in software hardening, and in code safety with no runtime support. It can be freely skipped by readers familiar with software exploitation. However, at the end of this section, we provide the threat model that is relevant to this article, which is important for all readers.

2.1 Exploiting Unsafe Code

Programs can be written in both safe and unsafe code. The latter form is much more attractive for systems software or for unrestricted, and often fast, execution with no constraints in accessing memory. For example, a web browser may realize an environment for executing scripts; such scripts are usually developed in JavaScript. Sometimes, the browser may choose to compile the script in native code at runtime. Compilation may involve low-level features, such as changing permissions of memory pages and injecting executable code on the fly. Thus, the browser is fairly dependent to this unrestricted memory access, and, of course, the browser is just an example of such software.

Unrestricted memory access may have security consequences, since an attacker may leverage vulnerabilities for forcing the program to arbitrarily overwrite (or overread) its own memory. The memory of the executing process may contain control data (for instance, return addresses or VTable pointers) or sensitive data (for instance, cryptographic keys). Changing control data or reading sensitive data can both degrade the security of the program. For example, by changing control data, an attacker can launch a control-flow attack and redirect the program to executing their code.

2.2 Hardening Unsafe Code

Software hardening aims at protecting code from memory corruption, without entirely losing the flexibility related to the aforementioned unrestricted memory-access model. Several techniques are enforced on top of unsafe systems with a goal to increase the difficulty of exploitation. As an example consider stack canaries [46]. These are random values injected in the stack of functions that contain local buffers. The random canary is copied from a hardware register to the stack, close to the return address. A linear overflow that aims at overwriting the return address, will also overwrite the canary, something that can be checked at the function epilogue (the program will crash upon inferring that the canary has been modified).

Notice that the canary does not change the memory-access model. The program is still unrestricted, and unsafe. A vulnerability can still take place and overwrite the stack. However, with a canary in place, a simple linear overflow is not enough anymore. The attacker needs to find new ways for bypassing this simple defense (leak the canary value, overwrite other control data, or perform the overwrite in a non-linear fashion). Beyond stack canaries, compilers, nowadays, implement advanced hardening techniques such as Control-flow Integrity (CFI) [2] and SafeStack [3].

2.3 Safe Code

Other options toward memory safety is constructing software that it is either memory safe at compile time or during runtime through a very limited runtime support. In the first case, the final binary may include checks injected by the compiler for preserving safety at runtime, however, there is no runtime support that takes certain decisions according to the execution of the program (e.g., such as a garbage collector). Constructing such programs may require radical changes in software engineering, since many common programming patterns are prohibited.

As an example, consider Rust, which enforces garbage collection without actually using runtime memory management. Rust assigns a tight scope to variables and prohibits compilation if multiple (writable) pointers point to a given memory address. This is important, since during deallocation of an object, Rust has ensured that only one reference points to it and, therefore, there are no concerns for violation of temporal safety [73]. The ideas behind this, which is generically known as *the borrow checker*, are central to the language design and are not covered in this article. For more information, we refer the reader to the official documentation [32] and other reviews of the programming language [42, 59].

Another example of such memory safe system is Go, which imposes memory safety by implementing a limited runtime support through a lightweight garbage collector [53]. An example of how Go provides memory safety is by implementing *bounds checks* [44] that are triggered during runtime. These checks ensure that the program will not behave abnormally by accessing invalid memory segments and thus, abusing spatial safety.

At this point, we just stress that programs with limited or no runtime support at all, are generically fast and can be easily combined with unsafe programs. For instance, a web browser implemented in C/C++ may outsource some of its functionality (usually, the one that is prone to software bugs, such as complex parsing, for example) to Rust code.

2.4 Threat Model

In this article, we assume software that has memory-corruption vulnerabilities (aligned with similar research [48, 57, 65, 70]), which may be transformed to powerful read and write primitives [73]. The particular software that we are interested in comes in the form of a *mixed* binary. This means that multiple compilers have been used to produce the final machine code, and each of these compilers follows a different memory model, as far as safety is concerned. As an example, we consider binaries that have been assembled by compiling C/C++ with no runtime management, and Rust, a programming system that has no runtime support, but imposes safety at compile time. One example of such a binary is Mozilla Firefox.

Furthermore, we assume that standard hardening techniques are in place, namely non-executable data [37], ASLR [66], and stack canaries [46], and, additionally, advanced hardening techniques that *only lately* have been enabled to standard compilers. These include CFI [2] and SafeStack [56] in Clang. As mentioned, we assume that software has memory-corruption vulnerabilities; however, *as software* we refer only to the mixed binary, and not to side products, such as the Rust compiler [21] or the CFI implementation in Clang.

The type of vulnerabilities we assume are standard memory-corruption bugs [73]. For instance, we build exploits for Mozilla Firefox based on CVE-2018-6126 [22], although similar vulnerabilities could serve our purpose.

3 HIGH-LEVEL ATTACK MECHANICS

In this section, we discuss the mechanics of the attack presented in this article. Later, in Section 7, we discuss the end-to-end proof-of-concept exploit for Mozilla Firefox. We begin by shortly reviewing the CFI [34] and SafeStack [3] support found in off-the-shelf industrial compilers, such as Clang. We then discuss some Rust basics, and how Rust can be mixed with unsafe code, and then we conclude with a toy example, which highlights how and why the attack works in principle.

3.1 Control-flow Integrity

One of the most promising hardening techniques is CFI, originally proposed almost one decade ago [34]. The technique suggests that a program is statically analyzed for estimating a legitimate CFG, which should be enforced at runtime, whenever indirect branches take place. In short, CFI

```
1    typedef void (*int_arg_fn)(int);
2
3    void func_int(int a){
4        ...
5    }
6
7    void func_float(float a){
8        ...
9    }
10
11   int main(int argc, char *argv[]){
12       int_arg_fn f = func_int;
13
14       bug();
15
16       /* This call is prevented by CFI. */
17       f();
18
19       return 0;
20   }
```

Fig. 1. Example of how Clang with CFI prevents a control-flow attack in C/C++ code. In `main`, a function pointer, of a compatible prototype, points to `func_int` (at line 12). Then the function `bug` is called, which, under the scenes, overwrites the function pointer to point to a non-compatible function, specifically to `func_float`. Once the function pointer is called (at line 17), the program crashes and prevents the function pointer from calling the non-compatible function.

```
1    void bug (void (**fpC) (int)) {
2        *fpC = (void(*)(int)) &target;
3    }
```

Fig. 2. The `bug()` function used to simulate the mechanics of the example attacks presented in this article. This function emulates an arbitrary write primitive acquired by an attacker. The attacker can write any memory address (input of the function) and set it to `target()` function of their choice. Typically, these memory addresses contain control data and the primitives are granted by abusing spatial or temporal safety [73]. This function is used only for conveying the mechanics of the example attacks. In Section 7, we use a real vulnerability (CVE-2018-6126 [22]) for delivering the actual PoC exploit.

computes all possible (legitimate) targets of an indirect branch. An attacker that overwrites control data used in an indirect branch is constrained to follow only the (legitimate) flows that were *a priori* computed. For instance, changing the control flow of a program to point to a ROP gadget [72] is not possible, since such flow will never be part of the computed CFG.

CFI has been realized in practice, especially for the forward edge [75] (i.e., for constraining the targets of function pointers and VTable-based calls in C++ programs), and it now ships with standard compilers, such as Clang [2]. In particular, as part of an upstream Clang future, CFI can protect *all* indirect branches or a subset of it (i.e., focus only on virtual calls). In *all* programs demonstrated in this article, we have compiled C/C++ code using the `-fsanitize=cfi` option (for enabling all CFI checks), `-fvisibility=hidden` (for additional CFI checks in classes without visibility attributes), and with Link-time Optimizations (i.e., `-flto`).

As a very short example, consider the code listed in Figure 1, which belongs to a C program, compiled with Clang and CFI enabled. Beyond the `main` function the program contains two functions, namely `func_int` and `func_float`, which take as an input an integer and a float, respectively. In `main`, a function pointer, of a compatible prototype, points to `func_int` (at line 12). Then, the function `bug` is called, which overwrites the function pointer to point to a non-compatible function, specifically to `func_float`. Once the function pointer is called (at line 17), the program crashes and prevents the function pointer from calling the non-compatible function.

Last, we briefly expand on the nature of the `bug` function (see Figure 2). We use this function in several examples of this article and its role is to emulate a vulnerability of the program. According to the threat model of this article (see Section 2.4), the attacker can write any memory

SafeStack-Disabled   **SafeStack-Enabled**

High-memory Addresses

Function Arguments
Return Address
Environment
Buffer_Rust

Rust

Function Arguments
Return Address
Environment
Buffer_Rust

Rust

Function Arguments
Return Address
%rbp (frame pointer)
Environment
Buffer_C

C

Function Arguments
Return Address
%rbp (frame pointer)
Environment

C

Buffer_C

Low-memory Addresses   SafeStack   Unsafe Stack
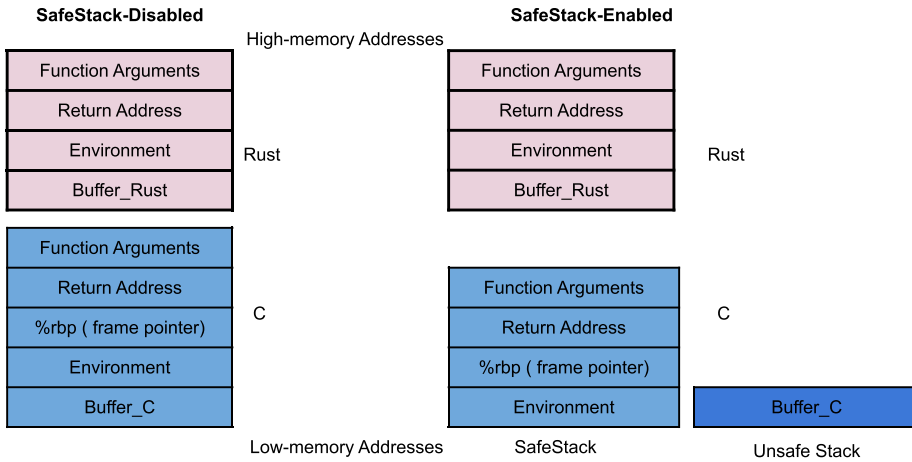
Fig. 3. Example of how SafeStack influences the stack frames of a Rust and a C function. In the SafeStack-disabled case, the stack frame of each function is stored in a single stack. In contrast, in the SafeStack-enabled case, the C function's return address is stored in the `safe` stack and its buffer in the `unsafe` stack. Nonetheless, since Rust is not influenced by SafeStack, its buffer remains in the same stack (`safe`) as before.

address (input of the function) and set it to a `target` function of their choice. Typically, these memory addresses contain control data and the primitives are granted by abusing spatial or temporal safety [73]. Notice, that several papers have employed similar bug functions [48, 57, 65, 70] to demonstrate their attacks.

3.2 SafeStack

SafeStack [3] is a hardening technique that suggests that a program's stack is separated into two different stacks, the safe and the unsafe one. The safe stack stores the return addresses and variables that are considered safe, meaning that they cannot be used for changing any control data, while everything else is stored in the unsafe stack. This way, a buffer overflow cannot be used to overwrite any control data like a function's return address, since these two are located in two different stacks. In *all* programs demonstrated in this article, we have compiled C/C++ code using the `-fsanitize=safe-stack` option for enabling SafeStack.

In Figure 3 we depict an example of how the stack is influenced when SafeStack is enabled for a stack frame of a Rust and a C function. In the SafeStack-disabled case, the stack frames of both Rust and C functions that store their return addresses and local variables, are located in the same stack. As a result, this could cause an unwanted behaviour if the return address of one of the functions gets overwritten by a buffer overflow. In contrast, in the SafeStack-enabled case, the C function's return address is stored in the `safe` stack and its buffer in the `unsafe` stack. Nonetheless, since Rust is not influenced by SafeStack, its buffer remains in the same stack (`safe`) as before and as a result, the control data are still vulnerable to a buffer overflow.

3.3 Rust Basics

Rust is a systems programming language developed by Mozilla with safety imposed at compile time. This essentially means that Rust has no heavy runtime support for ensuring that memory accessing does not violate safety. In contrast, Rust applies a different programming model where memory addresses have owners and ownership can be borrowed following very specific rules. The ideas behind this, which is generically known as *the borrow checker*, are central to the language

design and are not covered in this article. For more information, we refer the reader to the official documentation [32] and other reviews of the programming language [42, 59]. In this article, we only expand on particular features of Rust, which are central to the attacks we present, and these are (a) mixing Rust with C/C++ unsafe code (covered in Section 3.4), and (b) code constructs of Rust that force the compiler to emit indirect branches in the final binary (covered in Section 4).

3.4 Rust Calling Unsafe Code

All Rust code is considered safe, since otherwise the Rust compiler would have rejected the input. However, Rust supports calling C/C++ code, which generically is considered unsafe. For making this explicit, Rust provides the programmer with a very specific code block, called `unsafe { };`, for any direct calls to C/C++ code. Additionally, Rust provides certain directives for declaring non-Rust function prototypes. All this functionality is known as Foreign Function Interface (FFI) [30].

Nevertheless, Rust supports function pointers (as well as Closures and Traits, which we discuss in the next section). These function pointers can be used to call *indirectly* any code available in the executing process, no matter if this code has been produced by the Rust compiler or a C/C++ one. In fact, a Rust function pointer, if altered correctly, can point to anywhere, even to ROP gadgets [72], bypassing all CFI-based solutions, which normally require much more complicated exploitation mechanics [70].

To conclude, in the absence of bugs, there are two ways of calling unsafe code from Rust: (a) directly, using an `unsafe` block, and (b) indirectly, by using Rust function pointers, which point to (compatible) C/C++ functions. In all such cases, there is no warning emitted by the Rust compiler, as long as function pointers are compatible with the function prototypes that are allowed to point, legitimately, at compile time.

3.5 Toy-attack Example

CFI. In Figure 4 we depict an attack example in a program that contains code both written in C/C++ and Rust. The attack is based on modifying a Rust function pointer to call a C function with a *different* prototype compared to the original prototype as declared in the Rust code. In the upper part of the figure (lines 1–16) we depict the C/C++ part and in the bottom one (lines 1–10) the Rust part. We compile this program with Clang (CFI enabled) and the Rust compiler, producing zero warnings from both compilers. Once the final binary is run, all code is mapped to a *single* address space.

The C/C++ part (lines 1–16) contains two C functions, namely `func_int` and `func_float` (lines 4–10), exactly as the example of Figure 1, and additionally declares the prototype of a Rust function (lines 2 and 3), namely `frust`. This function's prototype takes two function pointers as arguments, one that can point to functions taking one integer as an input and one that can point to functions taking, further, a function pointer as an input. The Rust function is called in `main` (line 13) given two inputs: (a) the address of the `func_int` function, and (b) the address of the `bug` function. However, in the Rust part (lines 1–10), the Rust function `frust` will use the two input function pointers in the following way. The second function pointer will be called with the first function pointer as an argument. The result is that the `bug` function will change the first function pointer to point to a different (incompatible) function, namely `func_float`. As a result, `func_float` is called, indirectly, using a non compatible function pointer at line 9.

Observe, that the Rust function pointer can, legitimately, point to functions that take as input a single integer. Due to a memory-corruption bug, this pointer is used to call a function that takes a single *float*. A Rust compiler would have rejected a program that points a function pointer to a

```
1   /* C/Unsafe part.  */
2   extern void frust(void(*)(int),
3                         void(*)(void (**)(int)));
4   void func_int(int a){
5        ...
6   }
7
8   void func_float(float a){
9        ...
10  }
11
12  int main(int argc, char *argv[]){
13      frust(&func_int, &bug);
14
15      return 0;
16  }
```

```
1   /* Rust/Safe part.  */
2   #[no_mangle]
3   pub extern fn frust(f: fn(i32), bug: fn(&fn(i32))){
4       let a: i32 = 0 ;
5
6       bug(&f);
7
8       /* CFI should prevent this call.  */
9       f(a);
10  }
```

Fig. 4. Toy example of how Rust can attack hardened C/C++ code. In the upper part of the figure (lines 1–16) we depict the C/C++ part and in the bottom one (lines 1–10) the Rust part. Once compiled, all code is mapped to a single address space. The attack alters a Rust function pointer, through a memory-corruption vulnerability (see Figure 2), to point to a new target. Unlike all C/C++ function pointers that are CFI-protected, this pointer is not constrained and can be used to call any function (including ROP gadgets [72]).

non-compatible prototype, while a CFI-protected C/C++ program (see Figure 1) would have crashed when the overwritten function pointer is used.

This is the easiest attack discussed in this article; it is fairly artificially driven, but it encapsulates the core problem of all further (and more involved) attacks that we discuss in the next sections. In short, control data used by Rust code in indirect branches is not safeguarded, since the memory model used by Rust enforces both spatial and temporal safety at compile time. However, this memory model is not preserved when parts of code that adhere to a different memory model are present in the process space. And this can be devastating, since it is the non safeguarded control data produced by the Rust compiler that permits malicious control flows in the *hardened* executing process.

A final remark is that the particular control data that was modified (the Rust function pointer) was allocated in the stack of the program and, additionally, there was very specific interaction between the Rust and the C/C++ part. This is overly constrained, since performing the attack needs a very specific code pattern in place. It is questionable if real-world programs employ such coding style. From our study (see Section 6) this is not happening in Mozilla Firefox. However, we illustrate this example here just for understanding the root cause of the problem and alert developers who use particular features of Rust when the program is combined with unmanaged C/C++ code. In the next sections, we will deliver realistic attacks and we will ultimately overwrite VTable-like pointers, produced by the Rust compiler, stored on the heap of the process and influenced by C/C++ code, through memory corruption, and with *zero* interaction with the Rust part.

SafeStack. In Figure 5 we depict an attack example in a program that contains code both written in C/C++ and Rust. The attack is based on modifying the return address of a function written in Rust using a buffer overflow attack. In the upper part of the figure (lines 1–7), we depict the C/C++ part and in the bottom one (lines 1–9) the Rust part. We compile this program with Clang (SafeStack-enabled) and the Rust compiler, producing zero warnings from both compilers. Once the final binary is run, all code is mapped to a *single* address space.

```
1   /* C/Unsafe part. */
2   extern void frust(void(*)(char*));
3
4   int main(int argc, char *argv[]) {
5       frust(&bug);
6       return 0;
7   }
```

```
1   /* Rust/Safe part. */
2   use std::os::raw::c_char;
3
4   #[no_mangle]
5   pub extern fn frust(bug:fn(&[c_char;20])) {
6
7       let buffer:[c_char;20] = [0;20];
8       bug(&buffer);
9   }
```

Fig. 5. Toy example of how Rust can attack hardened C/C++ code. In the upper part of the figure (lines 1–7) we depict the C/C++ part and in the bottom one (lines 1–9) the Rust part. Once compiled, all code is mapped to a single address space. The attack alters the *frust* function return address through a buffer overflow attack, to point to a new target. Unlike all C/C++ buffers that are moved to the unsafe stack, the Rust buffer remains in the safe stack with all the control data.

The C/C++ part (lines 1–7) contains one C function, namely main (line 4) and additionally declares the prototype of Rust function (line 2), namely frust, which has a prototype that takes one function pointer that can point to a function taking a char buffer as an input. The Rust function is called in main (line 5) given one input, the address of the bug function. However, in the Rust part (lines 1–9), the Rust function frust will use the input function pointer to call the bug function with a c_char buffer as an argument created in frust. Then, the bug function will overwrite the return address of frust by overflowing the Rust buffer. As a result, frust returns to a malicious control flow.

Observe that the Rust function (frust) is calling a C function (bug) without the use of the unsafe block. A Rust compiler would have rejected a program that could potentially used for buffer overflow, while a SafeStack-protected C/C++ program would not let a modification of return addresses happen, since the buffers originated from C/C++ would be separated from all the control data.

4 ATTACK PRIMITIVES IN RUST

In Section 3 we have discussed how code of Rust and C/C++ can co-exist in the address space. We observed that Rust contains indirect branches that are not safeguarded and therefore memory corruption can leverage them for bypassing hardening, such as CFI. In this section, we further elaborate on other Rust programming concepts, which can provide the attacker similar primitives. In practice, the attacker, by means of memory corruption, seeks to influence indirect branches for introducing malicious control flows in the process, that are not contained by CFI.

4.1 Function Pointers

Programs that need to call a different function implementation at a particular call site may utilize function pointers. These are variables that can point to function entry points, can dynamically change during the process' lifetime, and can call a set of functions, depending on their value. Like C and C++, Rust supports function pointers, and they should be declared in advance following a particular prototype. For instance, f, below, is a function pointer that can point to any function that takes a single 32-bit integer as an input, and returns nothing.

```
f: fn(i32);
```

```
1    #[no_mangle]
2    pub extern fn frust(fp: fn(i32),
3                            bug: fn(&Fn(i32))) {
4        frust_cl(fp, bug);
5    }
6    pub extern fn frust_cl<F>(fp: F,
7                                bug: fn(&Fn(i32)))
8                                where F: Fn(i32) {
9        let a: i32 = 0;
10
11       bug(&fp);
12
13       fp(a);
14   }
```

Fig. 6. The attack presented in Section 3.5 using Rust closures instead of function pointers. This listing can be used to replace the safe part in Figure 4 if the vulnerable program contains closures instead of function pointers.

Function pointers are dangerous when memory corruption is possible, since a vulnerability can overwrite the value of a function pointer and force the program to call arbitrary code. Software hardening, such as CFI, can safeguard these pointers, so that they can, at least, point only to functions with a compatible signature.

As we discussed in Section 3, function pointers used in Rust are not safeguarded, since the memory model of Rust is immune to memory corruption. However, when Rust shares the same address space with C/C++, this is not true anymore. A memory-corruption error can change the value of a Rust function pointer and force the program to call arbitrary code.

4.2 Closures

Rust supports *closures* [25], which are functions that can capture their enclosing environment as well as inferring both their input and return types. For instance, `increment`, below is a closure that takes `i` as an input, increments it by `x` and returns it. In contrast with regular functions, closures can interact with the variables of their enclosing scope without receiving them as input, like in this case, `x`.

```
let x = 1;
let i = 1;
let increment = |i|   i + x ;
increment(i)
```

In Figure 6, we depict an example code that involves Rust closures. Function `frust` takes as input two function pointers, the second one takes as input an `Fn` [24] trait, which is used for taking a closure as an input parameter, in this case the function pointer received by `frust`. Next, in line 11 we assume that `frust` calls a C/C++ vulnerable function, namely `bug`, which alters the Rust `Fn` trait (`fp`), through a memory-corruption vulnerability to point to a new target. Finally, in line 13 the altered `fp` will call the new target.

4.3 Traits

Rust support a powerful concept for abstracting code, called *traits* [33]. By constructing certain traits, the programmer creates methods, which can be called by objects of different types. Rust's traits resemble very much the notion of *interfaces* that can be found in several object-oriented languages. One of the differences is that, through Rust traits, developers can create interfaces of existing types.

What is of interest in this article is the way traits are actually implemented by the Rust compiler and what the complications related to security and software exploitation are. For a thorough review of Rust traits, we refer the reader to the standard manuals [32].

```rust
1   #[no_mangle]
2   pub extern fn frust(){
3     let dog: Dog = Dog;
4     let cat: Cat = Cat;
5     let mut v: Vec<Box<Animal>> = Vec::new();
6
7     v.push(Box::new(cat));
8     v.push(Box::new(dog));
9     unsafe{
10      let x: usize = &v[1] as *const _ as usize;
11      bug(x);
12    }
13    for animal in v.iter() {
14      println!("{}", animal.make_sound());
15    }
16  }
17
18  trait Animal {
19    fn make_sound(&self) -> String;
20  }
21
22  struct Cat;
23  impl Animal for Cat {
24    fn make_sound(&self) -> String {
25      "meow".to_string()
26    }
27  }
28
29  struct Dog;
30  impl Animal for Dog {
31    fn make_sound(&self) -> String {
32      "woof".to_string()
33    }
34  }
```

Fig. 7. In lines 18–20, the trait, called `Animal`, is defined and supports a method called `make_sound` imple-
mented by `Cat` (lines 22–27) and `Dog` (lines 29–34). Furthermore, in lines 9–12 we assume that `frust` calls
a C/C++ vulnerable function, namely `bug`. The `bug` function, internally (not shown in the figure), corrupts
memory and when `make_sound` is called (line 14), control can reach any place the attacker likes. Exploita-
tion is possible due to the co-location of Rust and C/C++. A Rust-only program, equivalent to the one in
Figure 7, protects all (Rust) VTable pointers (at compile-time), while a hardened C/C++ program protects all
VTable pointers using CFI.

Rust traits can be implemented using static dispatch, pretty much like C++ templates, but also
using dynamic dispatch, which resembles very much VTable-based dynamic dispatch of virtual
methods in C++ [54]. For dynamic dispatch, a level of indirection is usually involved. Assume an
interface that converts an object to a string. All types that support this interface have to imple-
ment the particular method that converts them to a string, let us assume that this method is called
`to_string`. Now, consider a list that contains several objects of different type, but *all* of the types
implement the particular interface. Iterating the list and calling the conversion method for each
object involves calling a *completely* different method, but using the same naming conversion (i.e.,
call the `to_string` method for an object `x`, as long as `x` is of a type that supports the interface).
Compilers realize dynamic dispatch efficiently by putting all references to all different implemen-
tations of `to_string` in tables (usually called VTables) stored in read-only pages; objects allocated
in the heap or stack contain a pointer toward these tables. An iterator that hosts a variable that
calls `to_string` needs to simply be pointed to the right table, which holds the compatible imple-
mentation of the method.

In Figure 7, we depict an example code that involves Rust traits. In lines 18–20, the trait, called
`Animal`, is defined and supports a method called `make_sound`. Furthermore, in lines 22–27 and in
lines 29–34, two new types are defined, namely `Cat` and `Dog`, which both implement the `Animal`
trait. Now, in lines 2–16 there is the implementation of `frust` (in the same fashion we have done
in Figure 4 and Figure 6), which allocates a Rust vector (`Vec`), named `v`, and stores an instance of
`Cat` (`cat`) and an instance of `Dog` (`dog`). Both of these instances are allocated in the heap of the
program.

Furthermore, in lines 9–12 we assume that `frust` calls a C/C++ vulnerable function, namely `bug`. We use this form of interaction, here, for demonstration purposes, only. The actual PoC exploit we deliver for Mozilla Firefox in Section 7 does not contain such explicit form of interaction between Rust and C/C++. The `bug` function takes as an input the address of the last object pushed in the Rust vector (`dog`) and, internally (not shown in the figure), it corrupts memory and alters the VTable pointer stored in `dog`. Later in the code, in line 14, when `make_sound` is called, control can reach any place the attacker likes. We stress here that exploitation is possible due to the co-location of Rust and C/C++. A Rust-only program, equivalent to the one in Figure 7, protects all (Rust) VTable pointers (at compile-time), while a hardened C/C++ program protects all VTable pointers using CFI.

5 ATTACK PRIMITIVES IN GO

In Section 4, we have discussed Rust programming concepts that can provide the attacker with primitives, that can be used to introduce malicious control flows in the process. In this section, we discuss Go programming concepts, which can provide the attacker similar primitives to those observed in Rust. These programming concepts produce indirect branches that can be influenced by C/C++, since they are not safeguarded, for bypassing hardening.

5.1 Function Pointers

Similarly to Rust, C, and C++, Go also supports *function pointers*, namely variables that can point to a function entry point and change dynamically during a process' lifetime. These function pointers much like other types of variables, should be declared in advance before use, following a particular prototype. For instance, `fp`, below, is a function pointer that can point to any function that takes a single integer as an input and returns nothing.

```
var fp func(int)
```

As we discussed in Section 4, function pointers are dangerous when memory corruption is possible. Software hardening, such as CFI, can safeguard these pointers so that they can, at least, point only to functions with a compatible signature. However, like in Rust, functions pointers in Go are not safeguarded neither by hardening nor by its memory-model, and thus being susceptible to memory corruption.

In Figure 8, we depict an example code that involves Go function pointers. This is a toy example of how Go can attack hardened C/C++ code. In the upper part of the figure (lines 1–7) we depict the C/C++ part and in the bottom one (lines 1–17) the Go part. Furthermore, in line 14 we assume that `fgo` calls a C/C++ vulnerable function, namely `bug`. The `bug` function, internally (not shown in the figure), alters a Go function pointer, through a memory-corruption vulnerability (see Figure 2), to point to a new target, which is indirectly called in line 16 by the altered `fp`.

5.2 Closures

Go supports *closures*, which are a special case of anonymous functions or function literals. Closures can reference the variables of their enclosing environment [5] and interact with them without the need of receiving them as input. For instance, `sum`, below, is a function that takes a single integer as an input and returns a closure that both takes as input and returns, a single integer. Because the closure can access variables of its enclosing environment, it can access the variable `temp`, which holds the value of `x`. As a result, the anonymous function returns the sum of its input `y` and the value of `temp`.

```
1  /* C/Unsafe part.  */
2  extern void fgo(void(*)(void));
3
4  int main(int argc, char *argv[]){
5      fgo();
6      return 0;
7  }
```

```
1   /* Go/Safe part.  */
2   func func_int(a int){
3       ...
4   }
5
6   func func_float(a float64){
7       ...
8   }
9
10  //export fgo
11  func fgo () {
12      var a int = 0
13      var fp func(int) = func_int
14      C.bug()
15      /* CFI should prevent this call.  */
16      fp(a)
17  }
```

Fig. 8. Toy example of how Go can attack hardened C/C++ code. In the upper part of the figure (lines 1–7) we depict the C/C++ part and in the bottom one (lines 1–17) the Go part. Once compiled, all code is mapped to a single address space. The attack alters a Go function pointer, through a memory-corruption vulnerability (see Figure 2), to point to a new target. Unlike all C/C++ function pointers that are CFI-protected, this pointer is not constrained and can be used to call any function (including ROP gadgets [72]).

```
1   func sum(x int) func(int) int{
2       temp := x
3
4       C.bug()
5
6       return func(y int) int{
7           return temp + y
8       }
9   }
10
11  //export fgo
12  func fgo () {
13      fmt.Println(sum(5)(6))
14  }
```

Fig. 9. The attack presented in Section 5.1 using Go closures instead of function pointers. This listing can be used to replace the safe part in Figure 8 if the vulnerable program contains closures instead of function pointers. In line 13, fgo calls sum and passes the first argument. Furthermore, in line 4, we assume that sum calls a C/C++ vulnerable function, namely bug, which alters the Go indirect branch that would be used to call the anonymous function, through a memory-corruption error to point to a new target. Finally, in line 13, the altered indirect branch calls the new target.

```
func sum(x int) func(int) int{
    temp := x
    return func(y int) int{
        return temp + y
    }
}
```

In Figure 9, we depict an example code that involves Go closures. Function sum takes as input a single integer and returns an anonymous function that both takes as input and returns, a single integer. In line 4 we assume that sum calls a C/C++ vulnerable function, namely bug, which alters the Go indirect branch that would be used to call the anonymous function, through a

```
1   type Animal interface {
2       make_sound() string
3   }
4
5   type Cat struct {}
6   type Dog struct {}
7
8   func (c *Cat) make_sound() string {
9       return "meow"
10  }
11
12  func (d *Dog) make_sound() string {
13      return "woof"
14  }
15
16  //export fgo
17  func fgo () {
18
19      var v [2] Animal
20      v[0] = new(Cat)
21      v[1] = new(Dog)
22
23      C.bug()
24
25      for i := range v {
26          fmt.Println(v[i].make_sound())
27      }
28  }
```

Fig. 10. In lines 1–3, the interface, called `Animal`, is defined and supports a method called `make_sound` implemented by `Cat` (lines 8–10) and `Dog` (lines 12–14). Furthermore, in line 23 we assume that `fgo` calls a C/C++ vulnerable function, namely `bug`. The `bug` function, internally (not shown in the figure), corrupts memory and when `make_sound` is called (line 26), control can reach any place the attacker likes. Exploitation is possible due to the co-location of Go and C/C++. A Go-only program, equivalent to the one in Figure 10, protects all (Go) VTable pointers (at compile-time), while a hardened C/C++ program protects all VTable pointers using CFI.

memory-corruption error to point to a new target. Finally, in line 13, the altered indirect branch calls the new target, instead of the anonymous function.

5.3 Interfaces

Go supports *interfaces* [7], which are a set of method signatures that can be implemented through methods by `struct` types. When implemented, these methods can be called by objects of different `struct` type using both static and dynamic dispatch. For dynamic dispatch, a level of indirection is usually involved. Assume an interface that has a method signature that converts an object to a string. All `struct` types that support this interface have to implement this particular method that converts them to a string, let us assume that this method is called `make_sound`.

In Figure 10, we depict an example code that involves Go interfaces. In lines 1–3, the interface called *Animal* is defined and has a method signature called `make_sound`. Furthermore, in lines 8–10 and in lines 12–14, two new `struct` types are defined, namely *Cat* and *Dog*, which both implement the *Animal* interface. Now, in lines 17–28, there is the implementation of `fgo`, which defines an instance of `Cat` (`v[0]`) and an instance of `Dog` (`v[1]`). Both of these instances are allocated in the heap of the program and can be accessed by iterating the array `v`.

Furthermore, in line 23 we assume that `fgo` calls a C/C++ vulnerable function, namely `bug`. We use this form of interaction here for demonstration purposes only. The `bug` function corrupts memory and alters the VTable pointer stored in `dog`. Later in the code, in line 26, when `make_sound` is called, control can reach any place the attacker likes. We stress here that exploitation is possible due to the co-location of Go and C/C++. A Go-only program, equivalent to the one in Figure 10, protects all (Go) VTable pointers (at compile-time), while a hardened C/C++ program protects all VTable pointers using CFI.

```
1   type fn_int func() int
2
3   func foo_go(ch <-chan fn_int) {
4       fp := <-ch
5       C.bug()
6       fp()
7   }
8
9   func bar_go(ch chan<- fn_int) {
10      ch <- func() int {
11          return 0
12      }
13  }
14
15  //export fgo
16  func fgo() {
17      ch := make(chan fn_int, 2)
18      go foo_go(ch)
19      bar_go(ch)
20  }
```

Fig. 11. In line 17, the channel called ch is defined and can hold up to two function pointers, which can point to any function that receives nothing as input and returns a single integer. Next, in line 18, fgo calls foo_go using goroutines and passes the channel defined earlier as argument. Furthermore, in line 5 we assume that foo_go calls a C/C++ vulnerable function, namely bug. The bug function, internally (not shown in the figure), corrupts memory and when fp is called (line 6), control can reach any place the attacker likes. Exploitation is possible due to the co-location of Go and C/C++.

5.4 Channels

Go supports *channels* [4], which are pipes that enable the exchange of values between *goroutines* [6], namely threads that run concurrently. For instance, ch, below is a channel that can hold a single integer and uses the <- operator to receive a value from y and consequently send it to x.

ch := make(chan int)

ch <- y

x   := <- ch

In Figure 11 we depict an example code that involves Go channels. Function fgo defines a channel named ch, which can hold up to two function pointers, which in turn can point to any function that receives nothing as input and returns a single integer. Next, in line 18, fgo calls foo_go using goroutines and passes the channel defined earlier as argument. Function foo_go now runs in parallel with the thread of the main program. Then, in line 19, fgo calls bar_go and passes the channel defined earlier as argument. Furthermore, line 10, bar_go defines an anonymous function that passes it to ch. Once ch receives the anonymous function, it passes it to fp in line 4. Next, in line 5 we assume that foo_go calls a C/C++ vulnerable function, namely bug, which alters the Go indirect branch that would be used to call the anonymous function, through a memory-corruption vulnerability to point to a new target. Finally, in line 6 the altered fp will call the new target.

6 DISCOVERY OF PRIMITIVES

In this section, we discuss the methodology we use for finding attack primitives in Rust and Go code. Because this methodology is the same for the two programming languages, we elaborate only on the methodology for finding attack primitives in Rust code that ships with Mozilla Firefox. In practice, we need to analyze all Rust code and find patterns that encapsulate function pointers, closures, and traits, as discussed in Section 4. The methodology is based on two phases. First, we analyze the code statically, for finding interesting functions. These are functions that contain *indirect branches* as a result of using either Rust function pointers, closures, or traits. Then, we

dynamically analyze the code for inferring if identified indirect branches can be influenced by overwriting memory. Below, we discuss each of the two analysis phases in detail.

6.1 Static Analysis

Our static analysis methodology starts by locating all the functions that contain at least one indirect branch. To do that, we set the environment variable RUSTFLAGS="-emit asm" to extract the Rust code of the target program in assembly form. Notice, that this is the assembly produced by the Rust compiler and not the (approximated) output of a disassembler [38].

Then, we collect all the assembly files created from the above action and we run a python script that performs static analysis. Our script reads all the assembly files and creates data structures containing all the functions with their names and instructions. Afterward, it finds all the functions that contain at least one callq * instruction (e.g., callq *rax) that resembles an indirect branch in the disassembly. Then, the script produces a list of the names of the functions that were found containing at least one indirect branch.

A common incident we noticed during this analysis is that the majority of indirect branches are based on the value of the rip register. These branches are emitted by the Rust compiler for generating Position Independent Executable (PIE) code [67] for x86_64. This type of indirect branches is considered not exploitable, so we added an extra step for excluding such cases to make the dynamic analysis (discussed later) a lot faster. Given an identified indirect branch, the static-analysis script performs a backward analysis to determine if the target address of the branch originates from rip. If this is the case, then these indirect branches are marked and are excluded from the final function list produced.

The final output of the static analysis is a list of functions containing all their indirect branches, due to Rust function pointers, closures, and traits and not due to PIE code.

6.2 Dynamic Analysis

Static analysis produces a list of Rust functions that contain indirect branches. All these branches are vulnerable from C/C++ code; however, to attack them, the given input should be able to trigger the corresponding Rust code. For this, we use dynamic analysis to infer how the identified Rust code can be triggered.

Our dynamic analysis starts by reading the list of functions created by the static analysis. We implement dynamic analysis on top of gdb [27], so we run Mozilla Firefox with breakpoints to a number of functions from the aforementioned list. For faster analysis, we add breakpoints *only* to a set of identified functions and repeat this process until we add breakpoints to all the functions of the list. A debugging session with several breakpoints incurs significant overhead. For each session, Mozilla Firefox is directed to open a set of web pages for 10 minutes [14]. After each session is done, we record which breakpoints have been triggered.

During dynamic analysis, the script stops at breakpoints (Rust functions containing indirect branches) that are triggered by user input. When it reaches a breakpoint, the script steps in the execution, instruction by instruction, until it reaches an indirect branch. Then, it adds breakpoints to all the mov instructions that move the targeted function address to the register used in the indirect call (i.e., the dataflow reaching the register used in the indirect branch). The execution continues until it reaches these mov instructions and tries to determine the location of the target function address using the information located in the /proc/<pid>/maps, where <pid> is the process id of the analyzed Firefox. We do this because we want to locate all the indirect branches that are initialized by a heap address, which can be easily influenced by C/C++ code without

Table 1. Results of Static and Dynamic Analysis
as Performed in Mozilla Firefox

Analysis	#
functions	174,790
indirect branches	672,264
based on `rip`	669,073
not based on `rip`	3,191
no `drop_in_place`	2,755
Rust calls C	18

Observe that there are 174,790 Rust functions that contain more than 670,000 branches. However, most of them are emitted due to PIE code. If we deduct all branches, plus remove all branches contained in the unsafe `drop_in_place` function (used for object release), then we remain with 2,755 branches due to Rust function pointers, closures, and traits. Finally, only 18 indirect branches are produced when a Rust function calls indirectly a C function.

interacting with the Rust part. Finally, the output of the dynamic analysis is a list of functions with their indirect branches and the name and address space of the target function.

6.3 Results

We depict the results of the static and dynamic analysis in Table 1. Observe that there are 174,790 Rust functions that contain more than 670,000 branches. This is a vast amount of identified indirect branches, however, most of them are emitted due to PIE code [67]. These branches cannot be influenced by memory, since they just assist the executable code to be relocatable at runtime. If we deduct all these (relocation) branches, and if we remove all branches contained in the unsafe `drop_in_place` function (used for object release), then we remain with 2,755 branches. Now, *all* of these branches are contained in safe Rust code and are emitted due to the use of (Rust) function pointers, closures, and traits. The targets of these branches are all based on (control) data stored in the heap and the stack of the executing process. Several of these can be, easily, modified by C/C++ code (due to memory-corruption bugs) with *no* interaction between the C/C++ and Rust part. In addition, through dynamic analysis, we found that only 18 indirect branches are produced when a Rust function calls indirectly a C function.

Furthermore, in Table 2 we list all identified indirect branches in Rust as they are distributed in some major components. In particular, about 248 are found in Stylo (the CSS engine), 324 in WebRenderer (the now default rendering engine), and 438 in Crane (the WebAssembly compiler). Several more, about 1,745, are in other Rust code, included in the browser. Therefore, we conclude that these branches are in *core* functionality of the web browser, which can be triggered by user input.

7 EXPLOITING FIREFOX

In this section, based on our methodology outlined in Section 6, we exploit Mozilla Firefox 66.0a1 (2019-03-13, 64-bit) using CVE-2018-6126 [22]. We built Firefox with the following options written in the `mozconfig` file:

```
ac_add_options --disable-optimize
ac_add_options --enable-debug
```

Table 2. Indirect Branches in
Rust Are Distributed in Some
Major Components

Library	Volume
Stylo	248
WebRender	324
Crane	438
Other	1,745
Total	2,755

In particular, about 248 are found
in Stylo (the CSS engine), 324 in
WebRenderer (the now default ren-
dering engine), and 438 in Crane
(the WebAssembly compiler). Several
more, about 1,745, are in other Rust
code, included in the browser.

7.1 Challenges

Here we discuss the main challenges of the exploitation process.

(1) The main challenge is finding indirect branches in Rust, which are initiated with addresses
from the heap of the process. As we discussed in Section 6.3, despite the fact that there
are over 670,000 indirect branches in Rust, most of them are not exploitable (they are
related to PIE [67]), and those that are, can be loaded with addresses either from the stack
or heap. However, indirect branches that are loaded from the stack are much harder to
exploit, since they require a particular interaction between Rust and C/C++ code.

(2) Another common event we noticed is that many indirect branches are initialized and ex-
ecuted in a very tight fashion. An example of such case is the following.

```
1   0 x00007f f f e c220d8e <+3374>: mov %rdx, 0 x218(%r sp)
2   0 x00007f f f e c220d96 <+3382>: mov %r8,%rdx
3   0 x00007f f f e c220d99 <+3385>: mov 0x218(%r sp),%r8
4   0 x00007f f f e c220da1 <+3393>: cal lq *%r8
```

Notice that the above indirect branch is executed using r8; however, r8 is loaded with an address
in the previous instruction, and this address is overwritten just two instructions earlier, with the
contents of rdx. Now, influencing rdx is not always possible. Therefore, the memory used for
loading r8 cannot be overwritten, or it is very hard to be overwritten by a different thread.

7.2 Vulnerability

The vulnerability we use for delivering the exploit is based on CVE-2018-6126 [22]. The particu-
lar bug was originally reported for Mozilla Firefox 60.0.2. Therefore, we port the vulnerability to
Mozilla Firefox 66.0a1 (2019-03-13, 64-bit). Notice that reproducing actual CVEs is often fairly hard
and tedious [61] due to limited information, already applied patches and further code changes that
have occurred since the original posting of the bug. Porting real vulnerabilities to recent software,
using similar to real vulnerabilities or even entirely artificial bugs to demonstrate proof-of-concept
attacks is common to the research community [69, 70]. CVE-2018-6126 [22] is a typical heap buffer
overflow, which can be triggered by providing the browser with a specially crafted SVG image. De-
pending on the contents of the HTML document that contains the SVG image, the overflow can
be controlled and the pointer of the overflowed buffer can thus write to arbitrary memory. For

```
1    style :: media__queries :: media__feature__expression :: MediaFeatureExpression :: matches :
2
3    [ 26 instructions ]
4
5    mov     0x10(%rax),%rcx
6    mov     %rcx,0x108(%rsp)
7
8    [ 88 instructions ]
9
10   mov     0x108(%rsp),%rax
11
12   [ 6 instructions ]
13
14   callq   *%rax
```

Fig. 12. The skeleton of the Rust function that contains the indirect branch (line 14), used to exploit a hardened Firefox, and the dataflow reaching the branch (lines 5, 6, and 10).

understanding the mechanics and reproducing the CVE we heavily used Mozilla Firefox compiled with Address Sanitizer [71]. The PoC, of course, targets Mozilla Firefox with Address Sanitizer disabled. The PoC uses the CVE for both single-process and multiple-process Firefox builds [29].

7.3 PoC Exploit

Using the methodology outlined in Section 6, we know which Rust functions contain indirect branches and how these branches are loaded (either from heap or stack). For the particular PoC outlined here, we exploit an indirect branch contained in the Rust function `style::media_queries ::media_feature_expression::Media FeatureExpression::matches`, which is located at `servo/components/style/media_queries/media_feature_expression.rs`. The dataflow reaching the indirect branch contained in this function is depicted in Figure 12.

There are 62 functions called until our target function is called, the first 43 are C/C++ functions and the last 20 including our target function are Rust. For delivering the exploit we use CVE-2018-6126 [22], a typical heap buffer overflow, which can be triggered by providing the browser with a specially crafted SVG image. Depending on the contents of the HTML document that contains the SVG image, the overflow can be controlled and the pointer of the overflowed buffer can thus write to arbitrary memory.

Furthermore, the analysis of the targeted indirect branch reveals that it eventually calls `style::gecko:: media_features::eval_width`. The address of this method is loaded by reading the address `0x7fffd2e2d220` from Rust's heap. Using the arbitrary write (CVE-2018-6126 [22]), we re-write this heap address with the address of a new function that we control. Therefore, Firefox calls our function, instead of the legitimate target function, without crashing due to the CFI policy in place.

7.4 Failing to Exploit SafeStack

To bypass SafeStack, we need patterns like those depicted in Figure 5. Using the methodology explained in Section 6.2, we know which Rust functions contain indirect branches and whether the target function called indirectly is written in Rust or C/C++. We depict the results of the static and dynamic analysis in Table 1. Observe that there are only 18 indirect branches that display the pattern depicted in Figure 5, namely, a Rust function calling a C++ function indirectly. In short, bypassing SafeStack needs the specific pattern in which code written in Rust calls a C/C++ function, but as we showed in Table 1 there are very few cases with this pattern. As a result, SafeStack cannot be bypassed in practice. This is an experimental observation based on the results we have on Mozilla Firefox.

For a generic assessment, we need to analyze more mixed binaries. Since this is not possible due to the fact that all, except Mozilla Firefox, mixed binaries are closed source, at the moment,

we can only infer that SafeStack is much harder to bypass in practice. This means that the mixed binary should include very unique code patterns as shown earlier, that permit an attacker to bypass SafeStack. Notice, also, that Mozilla Firefox is not only a large project, in terms of analyzed code patterns, but also a reference project for integrating Rust with C/C++. We therefore anticipate that most projects will attempt to integrate Rust with C/C++ similarly to Mozilla Firefox.

8 DISCUSSION

In this section, we present a PoC exploit for a hardened mixed binary attacked through a Go library named `go-stats`. Furthermore, we argue that hardening safe code while being possible, is not a trivial process, since additional passes must be implemented for Rust and Go. In addition, we discuss several important points that stem from the PoC exploit presented in Section 7. Finally, we discuss the compile time and runtime checks for Rust, Go, and C/C++.

8.1 Go PoC Exploit

Using the methodology outlined in Section 6, we know which Go functions contain indirect branches. For the particular PoC outlined here, we exploit an indirect branch contained in the Go library `go-stats` [8]. This indirect branch is contained in the `go-stats.(*Stats).Write` function, which is located in `go-stats.stats.go`. To exploit this indirect branch, we created a mixed binary that combines Go and C/C++. The `go-stats` is invoked from within the part of the binary that is written in Go. Before `go-stats` is invoked, we firstly call a function located in C/C++ that emulates an arbitrary write primitive like is depicted in Figure 2. Through this function, we re-write the address of the indirect branch with the address of a new function that we control. As a result, the program calls our function, instead of the legitimate target function, without crashing due to the CFI policy in place.

8.2 Hardening Safe Code

One option to counter the attacks presented in this article is to extend hardening so that the whole process is protected. Notice that there are existing attempts toward hardening directly binaries, when source code is not available [76–78]. Therefore, hardening could be enabled in the overall mixed binary using one of those techniques, for instance TypeArmor [76]. We stress here that CFI hardening is based on computing correctly the CFG of a program; this computation can be done using the source code, the bitcode (LLVM) or just the machine code. Each option is a security tradeoff. An analysis closer to the original source code produces much more accurate results in terms of the computed CFG. In this article, we used binaries that utilize the CFI implementation as available in Clang/Clang++. For this particular implementation, the produced LLVM bitcode, after compiling a program, is carefully annotated to receive the CFI instrumentation during link-time optimizations. This annotation should be different if Rust or Go code is compiled instead of C/C++.

Therefore, we understand that extending CFI to the entire mixed binary is possible. Nevertheless, additional passes should be implemented for Rust and Go, and in particular for Go, since the original compiler is based on GCC and the LLVM-based one is less mature,[2] a compiler modification should be more appropriate [52]. These additional passes and compiler modifications should be further explored.

Finally, we stress that, at least, the Rust community has explored enabling some hardening techniques, but has done so only for the very basic ones [15, 16, 23], and not for any of the advanced ones available today [17, 19, 31].

---

[2]See discussion at https://golang.org/doc/faq#Do_Go_programs_link_with_Cpp_programs.
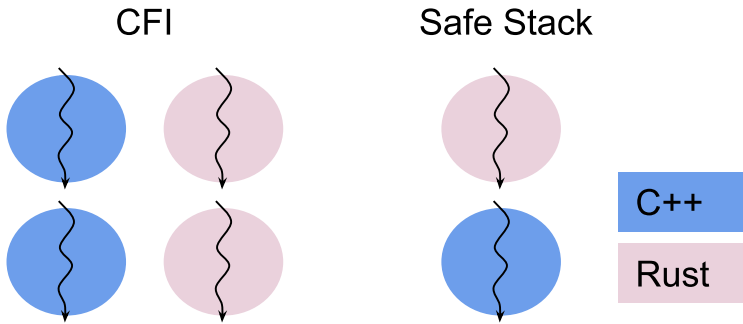
Fig. 13. CFI is vulnerable when Rust co-exists with C/C++, while SafeStack is practically hard to exploit. In the CFI case, there is no need for C/C++ and Rust interoperability; the vulnerable C/C++ code can execute and alter control data injected in the process image by Rust code. As a result, the exploit will work when the Rust code is eventually executed given that the function pointer is already overwritten by the C/C++ code. In contrast, for bypassing SafeStack, Rust code needs to explicitly call a C/C++ function with a stack vulnerability. Such code patterns are hard to find in existing software (e.g., Mozilla Firefox).

8.3 Hardening Bypass

The exploit uses Rust or Go code to easily bypass CFI, compared to generic CFI bypasses, such as COOP [70], where the attacker needs to locate particular COOP gadgets to launch the attack. While it is already proven that CFI can be bypassed [49], the attacks presented in this article are easier to perform. This is because the main prerequisite to execute these attacks is that the hardened code co-exists with a memory-safe system. In contrast, other attacks require the discovery of COOP gadgets that are included within the program's statically computed CFG, thus making it a cumbersome task.

Furthermore, the exploit *does not* work if Firefox is free of Rust, since CFI can contain the introduced malicious control flows. Nevertheless, it is the nature of CFI, as a hardening technique, that is vulnerable to Rust co-existing with C/C++ in the same address space. For instance, SafeStack [3] is shown not to be affected by the Rust code. The reason behind this, is the particular execution model, in terms of C/C++ and Rust executing code, required for the exploit to work. In the CFI case, there is no need for C/C++ and Rust interoperability; the vulnerable C/C++ code can execute and alter control data injected in the process image by Rust code. As a result, the exploit will work when the Rust code is eventually executed given that the function pointer is already overwritten by the C/C++ code. In contrast, for bypassing SafeStack, Rust code needs to explicitly call a C/C++ function with a stack vulnerability. Such code patterns are hard to find in existing software (e.g., Mozilla Firefox). Schematically, the difference between those two execution models is depicted in Figure 13.

8.4 Hardened Firefox

Throughout this article, we assume that Firefox is hardened using *standard* techniques available in Clang. However, this is not true with current Firefox releases. The source code is not compiled using CFI or SafeStack, as provided by Clang. However, we anticipate that in the coming years binaries will employ advanced hardening futures provided by compilers. Our anticipation stems, primarily, from: (a) the fact that deployment of past hardening techniques (stack canaries [46] and ASLR [66]) took a while, (b) big projects, such as Android, have started employing advanced hardening (based on CFI) by default [10], and (c) although deployment of mitigations can be slow, we are not hopeless, since numbers in hardening adoption are improving [74].

Table 3. Compile Time and Runtime Checks Performed
to Each Programming Language

Checks		Rust	Go	C/C++ with CFI
Compile time	Memory Safety	✓	✓	✗
	Bounds	✓	✓	✗
	Indirect Branches	✓	✓	✗
Run-time	Memory Safety	✗	✗	✗
	Bounds	✓	✓	✗
	Indirect Branches	✗	✗	✓

On the one hand, Rust/Go imposes memory safety mostly at compile time except with bounds checks in which it emits code that gets triggered during runtime. On the other hand, C/C++ programs with no hardening enabled, have no checks at all. However, when CFI is enabled, it ensures that the indirect calls are included within the program's statically computed CFG and thus preventing invalid control flows.

8.5 Compile Time - Runtime Checks

Throughout this article, we discuss software that is produced using multiple compilers that follow a different memory model, the memory-safe model and the memory-unsafe model. In this section we discuss their differences in terms of compile time and runtime checks.

In Table 3, we depict the various checks performed by each memory model during compile time and runtime. As we know, C/C++ does not perform any checks during compile time nor on runtime without any hardening enabled. However, when CFI is enabled, it ensures that the indirect calls are included within the program's statically computed CFG and thus, preventing any invalid control flows.

In contrast, Rust and Go perform several checks during compile time to ensure that there will not be any potential memory errors in the code. Nevertheless, observe that while both Rust and Go perform most of their checks during compile time, they also emit code that performs bounds checks during runtime, such as array bounds checks.

9 COUNTERMEASURES AND FUTURE WORK

The attack presented in this article directly affects mixed software. Until now, hardening techniques are capable of analysing programs that originate from a single programming language, usually C/C++ or Objective-C. Nevertheless, in this article we stress that there is a need for developing tools that can analyse mixed binaries or mixed code, in general. These tools should be able to handle programs that combine more than one high-level programming language, as an example consider the common case where C/C++ is mixed with Rust. Such tools can produce defenses that protect mixed software holistically and, at least, raise the bar for attacks like the one presented in this article.

In particular, our attacks focus on bypassing hardening techniques, such as CFI and SafeStack. The core weakness of the current implementations of these techniques is that they do not account for all source code that reaches the final binary. The analysis behind the instrumentation offered by CFI/SafeStack could be expanded for protecting parts outsourced to code that is not written in unsafe C/C++. Clearly, as already mentioned in this article, merging the analysis of Rust code with the analysis of the C/C++ code is not a trivial process, nevertheless, it is vital for securing the entire running process when C/C++ and Rust code are mixed.

Beyond, analysis for software hardening through instrumentation, like CFI/SafeStack, other defenses could be also assisted by analysis tools that target mixed software. Consider, for instance, memory allocators [35, 43, 62–64] and memory sanitizers [71]. These hardening techniques, again,

are based solely on the assumption that all the code is originally written in C/C++ and account only for unsafe allocations without considering allocations that are used by code that is written using a different system. In that case, a mixed-software hardened allocator could be realized by defining a single memory allocator that could coordinate the memory allocators of each language and have an overall picture of the whole program's data.

Finally, feedback-based fuzzing [1, 39, 40], where the target program is instrumented before being analyzed, can be also re-targeted for processing mixed binaries. Especially for fuzzing, the need to develop mixed-software analysis tools is important for performance, as well. Consider that, when fuzzing Mozilla Firefox, a significant part of Rust code should be completely avoided by the fuzzer, without of course omitting code that encapsulates interactions between C/C++ and Rust.

Therefore, analysis tools for mixed software can heavily assist in both countering attacks, like the one presented in this article, and improving other existing defense techniques based on hardened memory allocators, sanitizers and fuzzing tools. Clearly, such analysis tools cannot be produced by trivially extending the tools we currently have. We plan to explore the design and development of such tools in our future work.

10   RELATED WORK

Software hardening by means of CFI [34] has been realized in different flavors and levels. For instance, the granularity of CFI may vary and the instrumentation can be applied either at the source code (much more visibility, which affects the accuracy of computing the valid CFG) or at the binary level. It is true that several flavors of academic CFI-based prototypes have been bypassed in the past [45, 47, 49, 50]. Ultimately, even very fine-grained CFI techniques can be bypassed using counterfeit objects found in C++ [70] or Objective-C [57]. We stress here that all the aforementioned attacks are fairly sophisticated and questionable if they can always be carried out in practice with actual vulnerabilities found in the wild.

Today, CFI can be enabled at the compiler level, and in particular in Clang, while projects have started using the offered protection [10]. In this article, we do not claim a new attack for CFI, but we rather focus on easily nullifying CFI hardening when non-instrumented, but safe, code is also present in the process' address space. In the same spirit of this article, there have been published works that bypass CFI using non-instrumented code of JIT engines in web browsers [20, 41, 58]. In that sense, a memory snapshot of a browser that supports JIT compilation could be also classified as a *mixed* binary.

Beyond CFI hardening, today Clang also supports SafeStack [3], which is a subset of Code-pointer Integrity [56] and, in practice, prevents stack overflows from corrupting control data. This defense has a much more broad scope than stack canaries [46], which protect return addresses from linear overflows. The idea is to separate all unsafe buffers from control data that happens to be allocated in the stack (not just return addresses) [18]. CFI and SafeStack in Clang could be seen as complementary protections for securing both the forward and backward edge. SafeStack, like CFI, is also vulnerable [51], however, as we show in this article, non-instrumented Rust/Go code, does not pose any threat to the current implementation supported by Clang.

Finally, Rust and Go are core elements of this article. We stress that, for delivering the attack presented in this article, we do not rely on unsafe interactions between Rust/Go and C/C++ [36]. We rather exploit the fact that Rust/Go and C/C++ share the same address space but with *different* assumptions for the imposed memory model. Furthermore, we do not exploit weaknesses of techniques employed by the Rust compiler to infer safety at compile time. There are interesting works toward formally proving such guarantees [55, 68].

11 CONCLUSION

In this article, we consider *mixed* binaries, i.e., machine code that has been produced from different compilers and, in particular, from hardened C/C++ and Rust/Go. A prime example of such binary is Mozilla Firefox. Mixed binaries realize two different approaches toward reducing exploitation through memory corruption. Software hardening contains vulnerabilities during exploitation, on the other hand, Rust and Go prevent spatial/temporal vulnerabilities by checking the code, and enforcing certain programming patterns, at compile time or by utilizing a limited runtime support. Both software hardening and fast memory-safe systems, like Rust/Go, look promising directions for reducing software exploitation through memory corruption. Our research question, in this article, is if both approaches can be combined.

Toward answering the question, we argue that outsourcing certain functionality of an unsafe program to safe code may actually degrade the overall security of the program. Especially for hardened (unsafe) processes, the consequences may be negative, since vulnerabilities of the unsafe part, which otherwise could be contained by the hardening (e.g., CFI/SafeStack) involved, can be used for exploiting the running process. We demonstrated this through several artificial examples and two PoC exploits. The first PoC exploit is based on CVE-2018-6126 that can bypass CFI and attack the browser, through the Rust sub-part of Mozilla Firefox. The second one, attacks a hardened mixed binary and bypasses CFI through the Go library `go-stats`. Our attacks would not have been possible if machine code produced by the Rust/Go compiler was not present in the address space of the victim program. However, SafeStack is unlikely to be practically affected, since the code patterns needed for bypassing it are hard to find in existing software (e.g., Mozilla Firefox).

Finally, we delivered a tool that can scan Rust and Go software for identifying safe code that, if co-located with (hardened) unsafe code, can be leveraged to exploit vulnerabilities of the unsafe code, bypassing all hardening in place.

ARTIFACTS

The article is accompanied by the following artifacts, which are available at https://bitbucket.org/srecgrp/mixed-binaries-attacks-artifacts.

(1) Several PoC examples that combine C/C++ hardened code (with CFI and SafeStack) with Rust/Go code, and artificial exploits. The Rust programs come in the form of archives that are automatically built using the `cargo` tool and utilize function pointers, closures, and traits. The Go programs utilize function pointers, closures, interfaces and channels.

(2) Scripts that perform the static and dynamic analysis, as outlined in Section 6, in any version of Mozilla Firefox, as well as in any C/C++ software that is partially composed by Rust/Go. The scripts are written in Python.

(3) The PoC exploits, as outlined in Section 7 and in Section 8.1, for Mozilla Firefox 66.0a1 (2019-03-13) (64-bit) and `go-stats`, respectively.

ACKNOWLEDGMENTS

REFERENCES

[1] [n.d.]. American Fuzzy Lop. Retrieved November 2019 from http://lcamtuf.coredump.cx/afl.
[2] [n.d.]. Clang—Control flow integrity. Retrieved November 2019 from https://clang.llvm.org/docs/Control FlowIntegrity.html.
[3] [n.d.]. Clang - safestack. Retrieved November 2019 from https://clang.llvm.org/docs/SafeStack.html.
[4] [n.d.]. Go Channels. Retrieved May 2020 from https://tour.golang.org/concurrency/2.
[5] [n.d.]. Go Closures. Retrieved May 2020 from https://tour.golang.org/moretypes/25.

[6] [n.d.]. Go Goroutines. Retrieved May 2020 from https://tour.golang.org/concurrency/1.

[7] [n.d.]. Go Interfaces. Retrieved May 2020 from https://tour.golang.org/methods/9.

[8] [n.d.]. Go-stats. Retrieved May 2020 from https://github.com/segmentio/go-stats.

[9] [n.d.]. Golang. Retrieved May 2020 from https://golang.org/.

[10] [n.d.]. Kernel Control Flow Integrity. Retrieved November 2019 from https://source.android.com/devices/tech/debug/kcfi.

[11] [n.d.]. Mozilla Research—Rust. Retrieved from https://research.mozilla.org/rust/.

[12] [n.d.]. Rust in Production. Retrieved January 2020 from https://www.rust-lang.org/production.

[13] [n.d.]. Stack Overflow: Developer Survey Results 2020. Retrieved August 2020 from https://insights.stackoverflow.com/survey/2020.

[14] 2010. Kraken JavaScript Benchmark. Retrieved November 2019 from https://krakenbenchmark.mozilla.org/kraken-1.1/driver.html.

[15] 2014. Memory exploit mitigations #15179. Retrieved from https://github.com/rust-lang/rust/issues/15179.

[16] 2014. RFC: Memory exploit mitigation #145. Retrieved from https://github.com/rust-lang/rfcs/pull/145.

[17] 2014. Sanitize memory and CPU registers for sensitive data #17046. Retrieved from https://github.com/rust-lang/rust/issues/17046.

[18] 2015. Strengths and weaknesses of LLVM's safestack buffer overflow protection. Retrieved November 2019 from http://blog.includesecurity.com/2015/11/LLVM-SafeStack-buffer-overflowprotection.html.

[19] 2015. Update LLVM and add the safestack attribute to all generated functions. #26612. Retrieved from https://github.com/rust-lang/rust/issues/26612.

[20] 2017. Disarming control flow guard using advanced code reuse attacks. Retrieved March 2019 https://www.endgame.com/blog/technical-blog/disarming-control-flow-guard-using-advanced-code-reuse-attacks.

[21] 2017. Safe Rust code miscompilation due to a bug in LLVM's Global Value Numbering #45839. Retrieved March 2019 from https://github.com/rust-lang/rust/issues/45839, last accessed in March 2019.

[22] 2018. CVE-2018-6126: Heap buffer overflow rasterizing paths in SVG with Skia. Retrieved from https://bugzilla.mozilla.org/show_bug.cgi?id=1462682.

[23] 2018. Enabling Windows exploit mitigations by default in Rust programs? Retrieved March 2019 from https://internals.rust-lang.org/t/enabling-windows-exploit-mitigations-by-default-in-rust-programs/8716.

[24] 2019. As input parameters. Retrieved November 2019 from https://doc.rust-lang.org/rust-by-example/fn/closures/input_parameters.html.

[25] 2019. Closures. Retrieved November 2019 from https://doc.rust-lang.org/rust-by-example/fn/closures.html.

[26] 2019. Community makes Rust an easy choice for npm. Retrieved January 2020 from https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf.

[27] 2019. GDB: The GNU Project Debugger. Retrieved November 2019 from https://www.gnu.org/software/gdb.

[28] 2019. Implications of rewriting a browser component in rust. Retrieved from https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/.

[29] 2019. Multiprocess Firefox. Retrieved from https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Multiprocess_Firefox.

[30] 2019. Rust—Foreign function interface. Retrieved November 2019 from https://doc.rust-lang.org/nomicon/ffi.html.

[31] 2019. Rust 2019: Security. Retrieved from https://snf.github.io/2019/01/10/rust-2019-security/.

[32] 2019. The rust programming language. Retrieved November 2019 from https://doc.rust-lang.org/book/.

[33] Aaron Turon. 2019. Abstraction without overhead: Traits in rust. Retrieved November 2019 from https://blog.rust-lang.org/2015/05/11/traits.html.

[34] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'05)*. 340–353.

[35] Periklis Akritidis. 2010. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the USENIX Security Symposium (USENIX SEC'10)*. 177–192.

[36] Hussain M. J. Almohri and David Evans. 2018. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY'18)*. ACM, New York, NY, 248–255. DOI:https://doi.org/10.1145/3176258.3176330

[37] Starr Andersen and Vincent Abella. 2004. Changes to functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory protection technologies, data execution prevention. *Microsoft TechNet Library*. Retrieved from http://technet.microsoft.com/en-us/library/bb457155.aspx.

[38] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the 25th USENIX Security Symposium (USENIX SEC'16)*.

[39] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*.

[40] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with input-to-state correspondence. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*.

[41] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. 2015. The devil is in the constants: Bypassing defenses in browser JIT engines. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*. The Internet Society.

[42] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. 2017. System programming in rust: Beyond safety. *SIGOPS Oper. Syst. Rev.* 51, 1 (September 2017), 94–99. DOI:https://doi.org/10.1145/3139645.3139660

[43] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI'06)*. 158–168.

[44] Vincent Blanchon. [n.d.]. Go: Memory Safety with Bounds Check. Retrieved May 2020 from https://medium.com/a-journey-with-go/go-memory-safety-with-bounds-check-1397bef748b5.

[45] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the USENIX Security Symposium (USENIX SEC'15)*. 161–176.

[46] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, et al. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, Vol. 81. 346–355.

[47] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the USENIX Security Symposium (USENIX SEC'14)*. 401–416.

[48] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, New York, NY, 901–913. DOI:https://doi.org/10.1145/2810103.2813646

[49] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'15)*. 901–913.

[50] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out Of control: Overcoming control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'14)*. 575–589.

[51] Enes Goktas, Angelos Oikonomopoulos, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Bypassing clang's safestack for fun and profit. In *Proceedings of the Black Hat Europe Conference (Black Hat Europe'16)*.

[52] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. ShrinkWrap: VTable protection without loose ends. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'15)*. ACM, 341–350.

[53] Rick Hudson. 2018. Getting to go: The journey of go's garbage collector. Retrieved May 2020 from https://blog.golang.org/ismmkeynote.

[54] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing c++ virtual calls from memory corruption attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society.

[55] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, Article 66 (December 2017), 34 pages. DOI:https://doi.org/10.1145/3158154

[56] Kuznetsov, Volodymyr and Szekeres, László and Payer, Mathias and Candea, George and Sekar, R. and Song, Dawn. 2014. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 147–163. http://dl.acm.org/citation.cfm?id=2685048.2685061

[57] Julian Lettner, Benjamin Kollenda, Andrei Homescu, Per Larsen, Felix Schuster, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Michael Franz. 2016. Subversive-C: Abusing and protecting dynamic message dispatch. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'16)*. 209–221.

[58] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2017. Dachshund: Digging for and securing (non-)blinded constants in JIT code. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*.

[59] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The rust language. *Ada Lett.* 34, 3 (October 2014), 103–104. DOI:https://doi.org/10.1145/2692956.2663188

[60] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications (OOPSLA'13)*. ACM, New York, NY, 1–18. DOI:https://doi.org/10.1145/2509136.2509515

[61] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, Baltimore, MD, 919–936.

[62] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Not.* 44, 6 (2009), 245–258.

[63] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: Compiler-enforced temporal safety for C. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM'10)*. 31–40.

[64] Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the heap. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'10)*. 573–584.

[65] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking holes in information hiding. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*. USENIX Association, Berkeley CA, 121–138.

[66] PaX Team. 2003. Address Space Layout Randomization (ASLR). Retrieved from http://pax.grsecurity.net/docs/aslr.txt.

[67] Mathias Payer. 2012. Too much PIE is bad for performance. *Technical Report* 766 (2012).

[68] Eric Christopher Reed. 2015. Patina: A formalization of the rust programming language. *Master's Thesis. University of Washington* (2015).

[69] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*.

[70] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*.

[71] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12)*. 309–318.

[72] Hovav Shacham. 2007. 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*. 552–61.

[73] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P'13)*. IEEE Computer Society, Los Alamitos, CA, 48–62. DOI: https://doi.org/10.1109/SP.2013.13

[74] Theofilos Petsios. 2019. Millions of Binaries Later: A Look Into Linux Hardening in the Wild. Retrieved November 2019 from https://capsule8.com/blog/millions-of-binaries-later-a-look-into-linux-hardening-in-the-wild/.

[75] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the USENIX Security Symposium (USENIX SEC'14)*. 941–955.

[76] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough `call`: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'16)*. 934–953.

[77] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. 2013. Practical control flow integrity & randomization for binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'13)*. 559–573.

[78] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the USENIX Security Symposium (USENIX SEC'13)*. 337–352.