When to choose Rust

Tim McNamara
Senior Software Engineer, AWS New Zealand Ltd

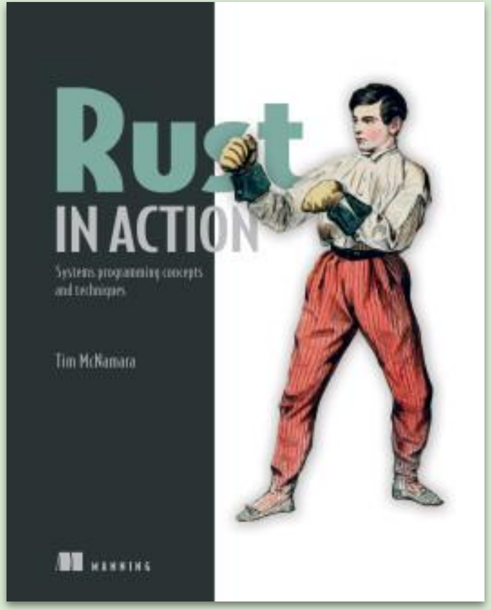
https://twitter.com/timClicks
https://youtube.com/timClicks
https://linkedin.com/in/timmcnamaranz

An acknowledgement

I’d like to begin by acknowledging the Noongar people, as traditional owners of South Western Australia land where we meet today. I would also like to pay my respects to Elders past, present, and emerging.

```
$ whoami

@timClicks
```



Rust IN ACTION

Systems programming concepts and techniques

Tim McNamara

MANNING

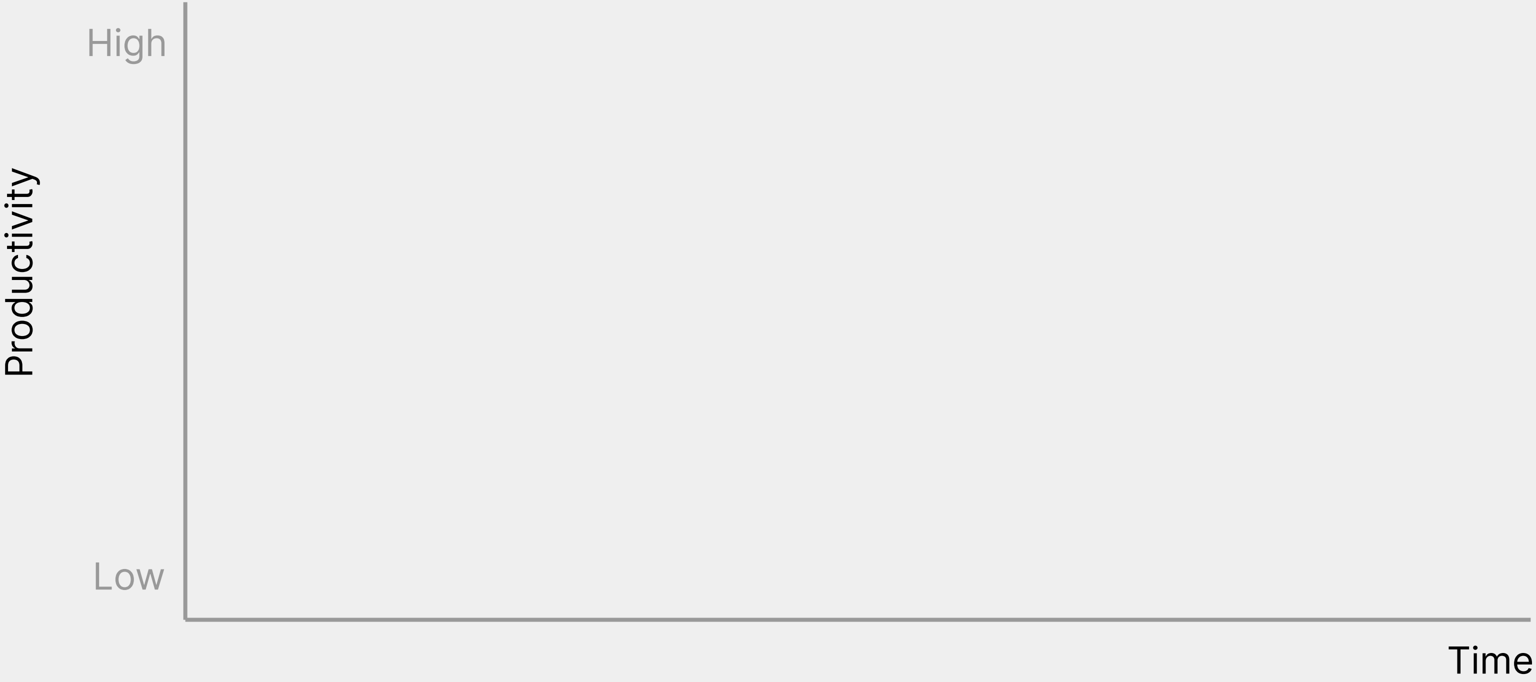
https://manning.com/mcnamara

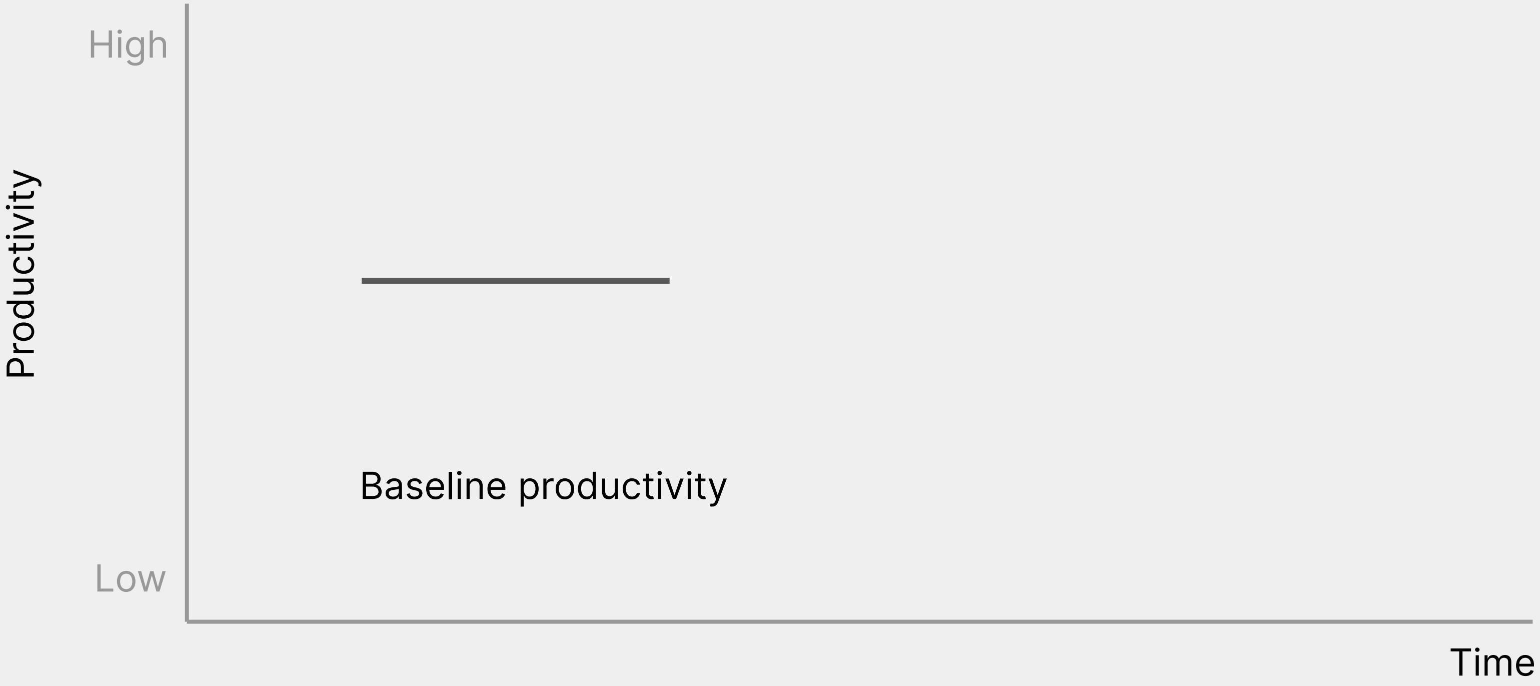
Aim for the talk

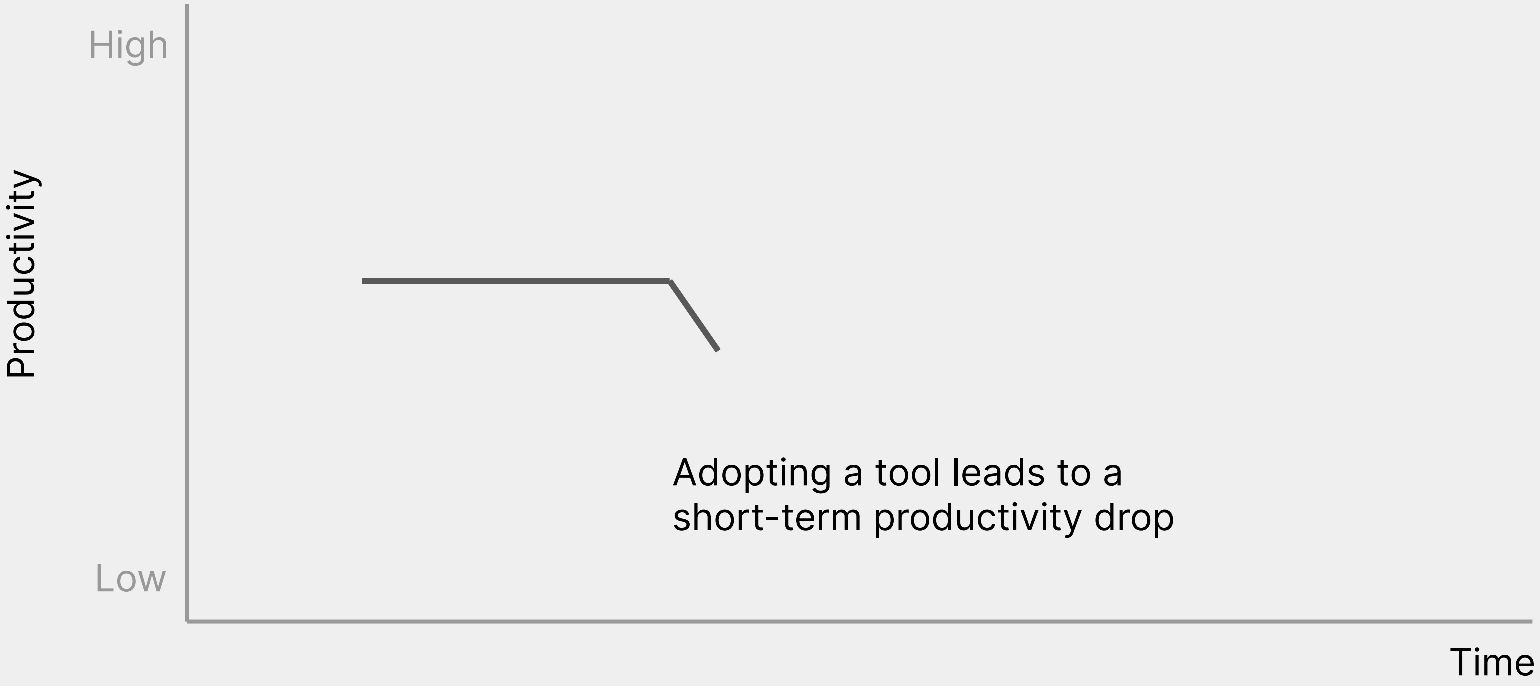
Convince you that Rust is worth evaluating, not that Rust is what your conclusion should be.

Your decision

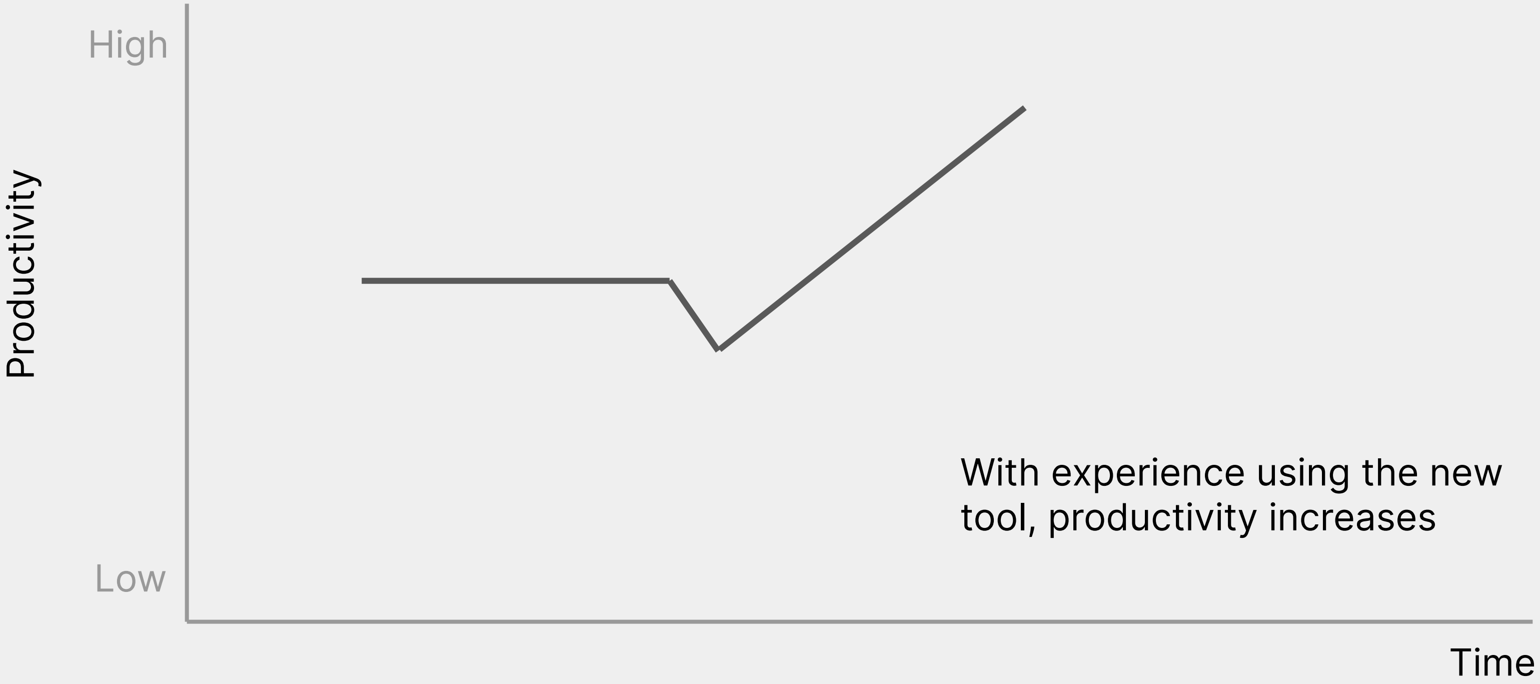
Are the long-term benefits of Rust sufficient to outweigh short-term costs?


High
Productivity
Low
Time

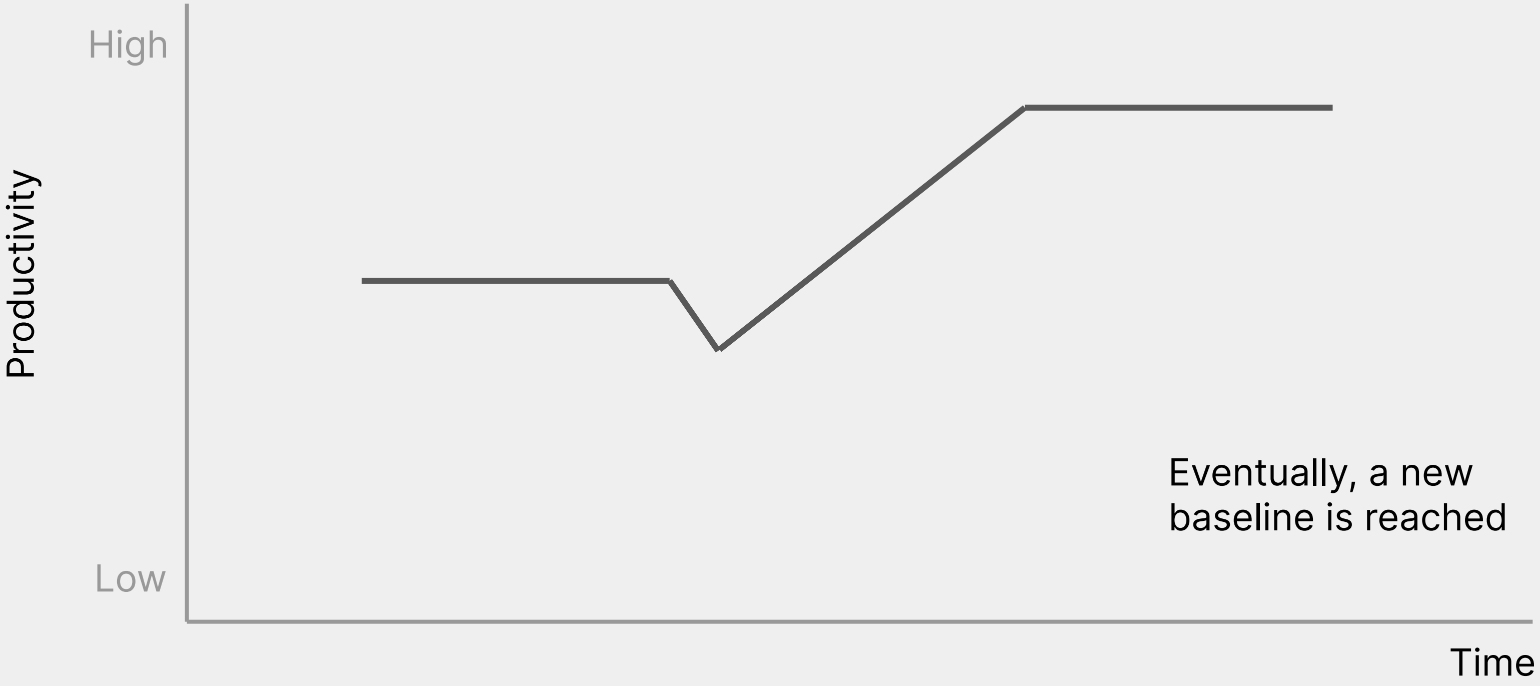

High
Productivity
Baseline productivity
Low
Time



Adopting a tool leads to a short-term productivity drop



With experience using the new tool, productivity increases

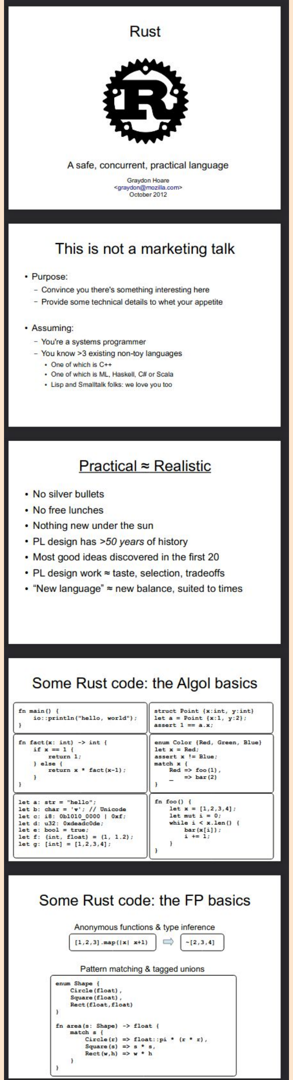


Eventually, a new baseline is reached

What is Rust?

Rust emerged from the Cyclone project, which was unable to bolt on safety to C.

It was envisaged as a practical programming language with no novel features - everything in the language should already be well known in research languages.



Rust

A safe, concurrent, practical language

Graydon Hoare
<graydon@mozilla.com>
October 2012

This is not a marketing talk

- Purpose:
  - Convince you there's something interesting here
  - Provide some technical details to whet your appetite
- Assuming:
  - You're a systems programmer
  - You know >3 existing non-toy languages
    - One of which is C++
    - One of which is ML, Haskell, C# or Scala
    - Lisp and Smalltalk folks: we love you too

Practical ≈ Realistic

- No silver bullets
- No free lunches
- Nothing new under the sun
- PL design has *>50 years* of history
- Most good ideas discovered in the first 20
- PL design work ≈ taste, selection, tradeoffs
- "New language" ≈ new balance, suited to times

Some Rust code: the Algol basics

```
fn main() {
    io::println("hello, world");
}
```

```
struct Point {x:int, y:int}
let a = Point {x:1, y:2};
assert 1 == a.x;
```

```
fn fact(x: int) -> int {
    if x == 1 {
        return 1;
    } else {
        return x * fact(x-1);
    }
}
```

```
enum Color {Red, Green, Blue}
let x = Red;
assert x != Blue;
match x {
    Red => foo(1),
    _   => bar(2)
}
```

```
let a: str = "hello";
let b: char = 'v'; // Unicode
let c: i8: 0b1010_0000 | 0xf;
let d: u32: 0xdeadc0de;
let e: bool = true;
let f: (int, float) = (1, 1.2);
let g: [int] = [1,2,3,4];
```

```
fn foo() {
    let x = [1,2,3,4];
    let mut i = 0;
    while i < x.len() {
        bar(x[i]);
        i += 1;
    }
}
```

Some Rust code: the FP basics

Anonymous functions & type inference

```
[1,2,3].map(|x| x+1)
```

⇨

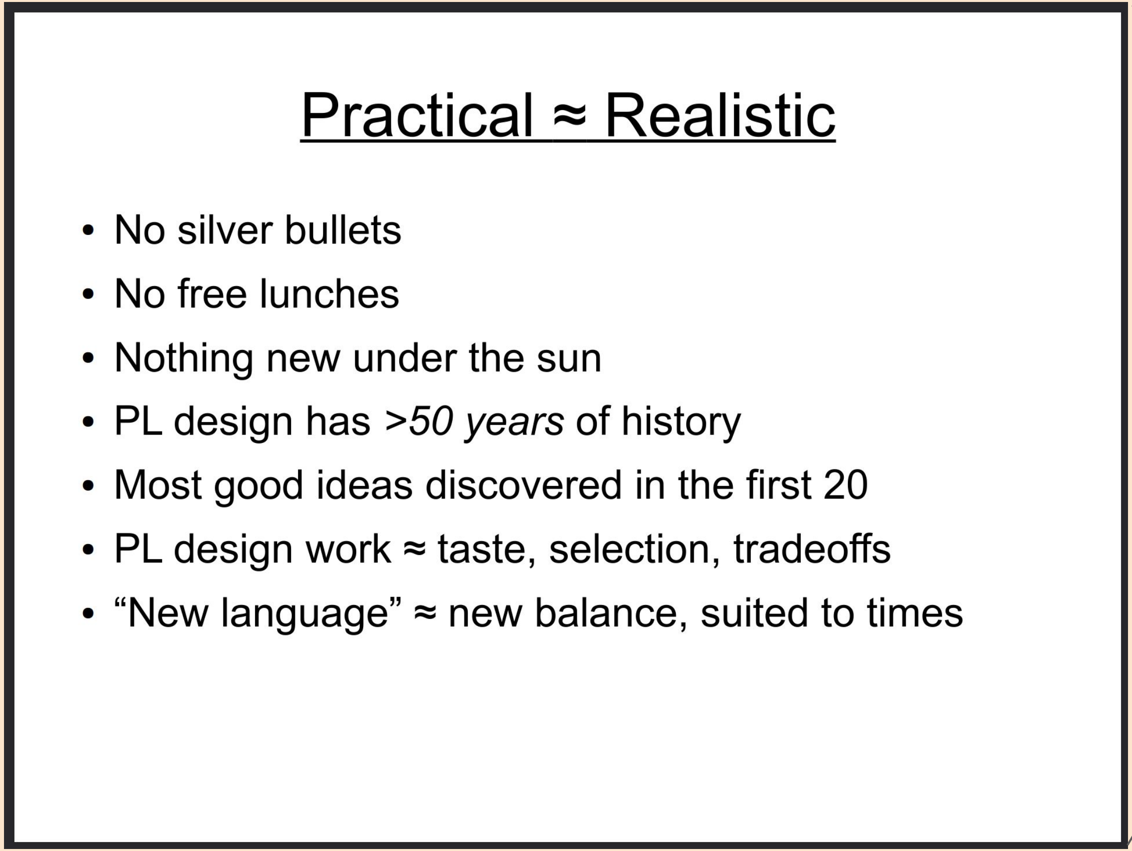
```
~[2,3,4]
```

Pattern matching & tagged unions

```
enum Shape {
    Circle(float),
    Square(float),
    Rect(float,float)
}

fn area(s: Shape) -> float {
    match s {
        Circle(r) => float::pi * (r * r),
        Square(s) => s * s,
        Rect(w,h) => w * h
    }
}
```

http://venge.net/graydon/talks/rust-2012.pdf



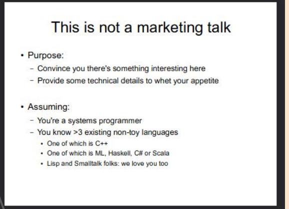
Practical ≈ Realistic

- No silver bullets
- No free lunches
- Nothing new under the sun
- PL design has *>50 years* of history
- Most good ideas discovered in the first 20
- PL design work ≈ taste, selection, tradeoffs
- “New language” ≈ new balance, suited to times

Rust

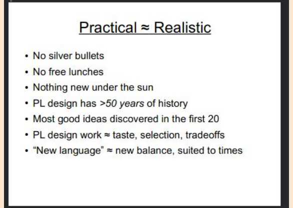
A safe, concurrent, practical language

Graydon Hoare
<graydon@mozilla.com>
October 2012



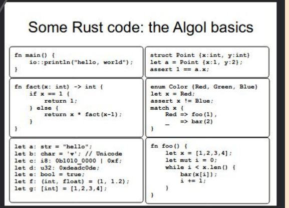
This is not a marketing talk

- Purpose:
  - Convince you there's something interesting here
  - Provide some technical details to whet your appetite
- Assuming:
  - You're a systems programmer
  - You know >3 existing non-toy languages
    - One of which is C++
    - One of which is ML, Haskell, C# or Scala
    - Lisp and Smalltalk folks: we love you too



Practical ≈ Realistic

- No silver bullets
- No free lunches
- Nothing new under the sun
- PL design has *>50 years* of history
- Most good ideas discovered in the first 20
- PL design work ≈ taste, selection, tradeoffs
- “New language” ≈ new balance, suited to times



Some Rust code: the Algol basics

```
fn main() {
    io::println("hello, world");
}
```

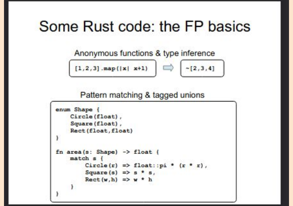
```
struct Point {x:int, y:int}
let a = Point {x:1, y:2};
assert 1 == a.x;
```

```
fn fact(x: int) -> int {
    if x == 1 {
        return 1;
    } else {
        return x * fact(x-1);
    }
}
```

```
enum Color {Red, Green, Blue}
let x = Red;
assert x != Blue;
match x {
    Red => foo(1),
    _   => bar(2)
}
```

```
let a: str = "hello";
let b: char = 'v'; // Unicode
let c: i8: 0b1010_0000 | 0xf;
let d: u32: 0xdeadc0de;
let e: bool = true;
let f: (int, float) = (1, 1.2);
let g: [int] = [1,2,3,4];
```

```
fn foo() {
    let x = [1,2,3,4];
    let mut i = 0;
    while i < x.len() {
        bar(x[i]);
        i += 1;
    }
}
```



Some Rust code: the FP basics

Anonymous functions & type inference

```
[1,2,3].map(|x| x+1)
```

⇨

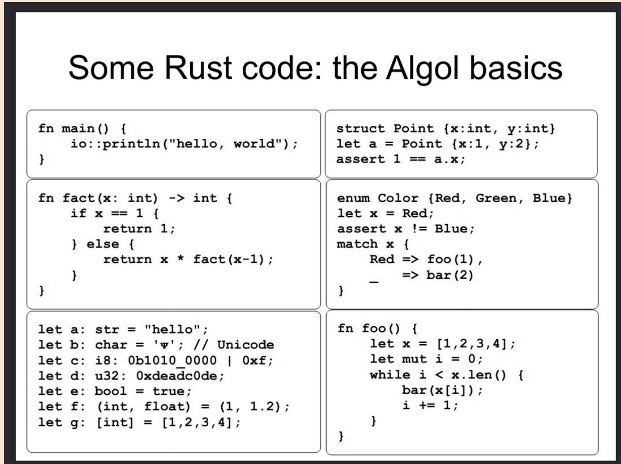
```
~[2,3,4]
```

Pattern matching & tagged unions

```
enum Shape {
    Circle(float),
    Square(float),
    Rect(float,float)
}

fn area(s: Shape) -> float {
    match s {
        Circle(r) => float::pi * (r * r),
        Square(s) => s * s,
        Rect(w,h) => w * h
    }
}
```

http://venge.net/graydon/talks/rust-2012.pdf



Some Rust code: the Algol basics

```
fn main() {
    io::println("hello, world");
}
```

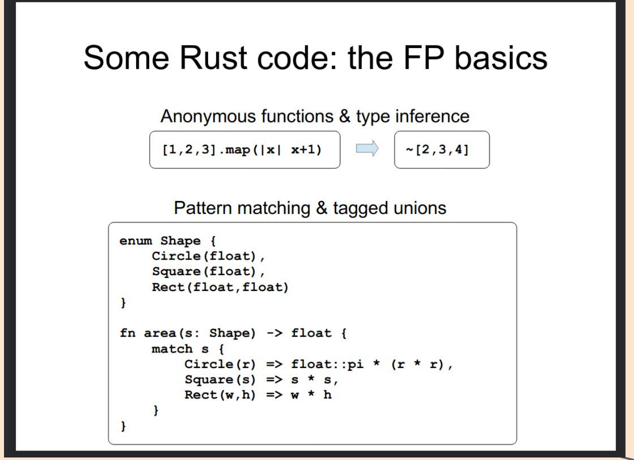
```
struct Point {x:int, y:int}
let a = Point {x:1, y:2};
assert 1 == a.x;
```

```
fn fact(x: int) -> int {
    if x == 1 {
        return 1;
    } else {
        return x * fact(x-1);
    }
}
```

```
enum Color {Red, Green, Blue}
let x = Red;
assert x != Blue;
match x {
    Red => foo(1),
    _   => bar(2)
}
```

```
let a: str = "hello";
let b: char = 'ψ'; // Unicode
let c: i8: 0b1010_0000 | 0xf;
let d: u32: 0xdeadc0de;
let e: bool = true;
let f: (int, float) = (1, 1.2);
let g: [int] = [1,2,3,4];
```

```
fn foo() {
    let x = [1,2,3,4];
    let mut i = 0;
    while i < x.len() {
        bar(x[i]);
        i += 1;
    }
}
```



Some Rust code: the FP basics

Anonymous functions & type inference

```
[1,2,3].map(|x| x+1)
```

⇨

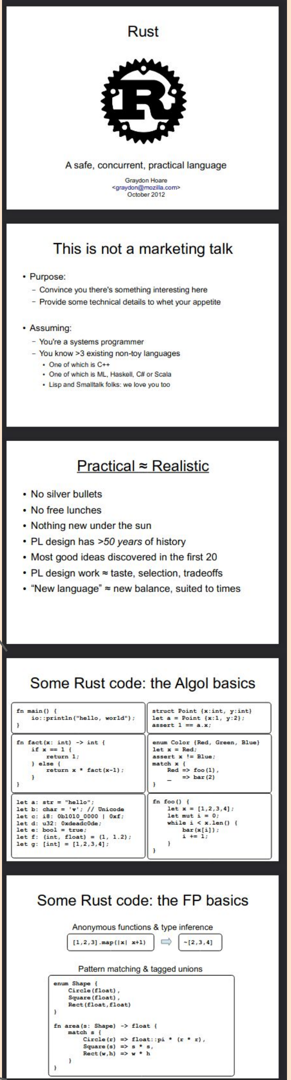
```
~[2,3,4]
```

Pattern matching & tagged unions

```
enum Shape {
    Circle(float),
    Square(float),
    Rect(float,float)
}

fn area(s: Shape) -> float {
    match s {
        Circle(r) => float::pi * (r * r),
        Square(s) => s * s,
        Rect(w,h) => w * h
    }
}
```

http://venge.net/graydon/talks/rust-2012.pdf



Rust

A safe, concurrent, practical language

Graydon Hoare
<graydon@mozilla.com>
October 2012

This is not a marketing talk

- Purpose:
  - Convince you there's something interesting here
  - Provide some technical details to whet your appetite
- Assuming:
  - You're a systems programmer
  - You know >3 existing non-toy languages
    - One of which is C++
    - One of which is ML, Haskell, C# or Scala
    - Lisp and Smalltalk folks: we love you too

Practical ≈ Realistic

- No silver bullets
- No free lunches
- Nothing new under the sun
- PL design has >*50 years* of history
- Most good ideas discovered in the first 20
- PL design work ≈ taste, selection, tradeoffs
- "New language" ≈ new balance, suited to times

Some Rust code: the Algol basics

```
fn main() {
    io::println("hello, world");
}
struct Point {x:int, y:int}
let a = Point {x:1, y:2};
assert 1 == a.x;
fn fact(x: int) -> int {
    if x == 1 {
        return 1;
    } else {
        return x * fact(x-1);
    }
}
enum Color {Red, Green, Blue}
let x = Red;
assert x != Blue;
match x {
    Red => foo(1),
    _   => bar(2)
}
let a: str = "hello";
let b: char = 'ψ'; // Unicode
let c: i8: 0b1010_0000 | 0xf;
let d: u32: 0xdeadc0de;
let e: bool = true;
let f: (int, float) = (1, 1.2);
let g: [int] = [1,2,3,4];
fn foo() {
    let x = [1,2,3,4];
    let mut i = 0;
    while i < x.len() {
        bar(x[i]);
        i += 1;
    }
}
```

Some Rust code: the FP basics

Anonymous functions & type inference

```
[1,2,3].map(|x| x+1)
```

⇨

```
~[2,3,4]
```

Pattern matching & tagged unions

```
enum Shape {
    Circle(float),
    Square(float),
    Rect(float,float)
}

fn area(s: Shape) -> float {
    match s {
        Circle(r) => float::pi * (r * r),
        Square(s) => s * s,
        Rect(w,h) => w * h
    }
}
```

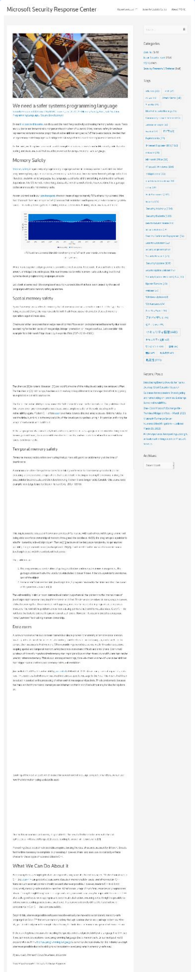
Why Rust?

Why Rust?

Users deserve
safe, secure software

(They are people, after all)

Unfortunately,
the very best
programmers
are not able to write
safe, secure software



Microsoft Security Response Center

We need a safer systems programming language

Memory Safety

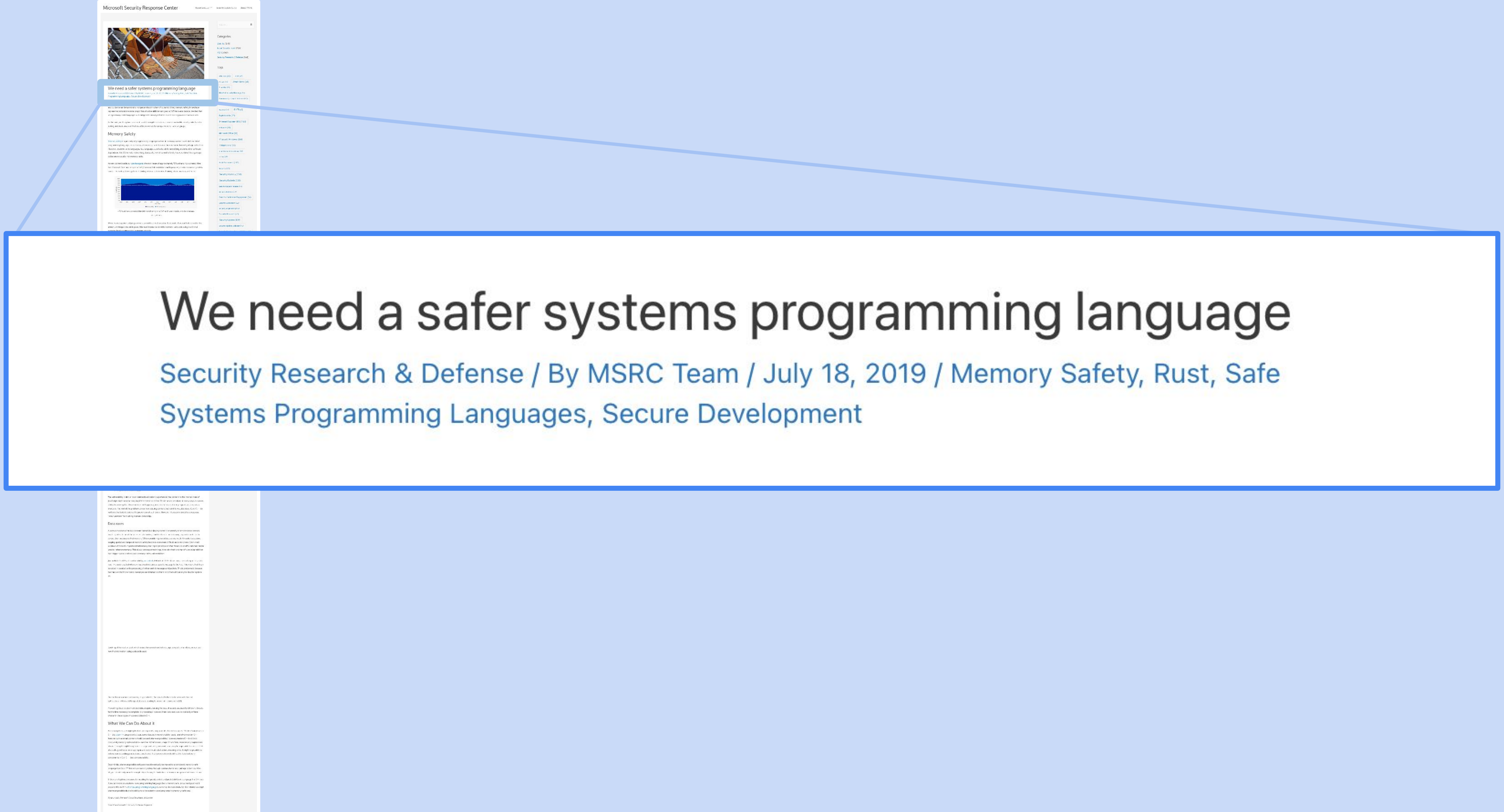
Spatial memory safety

Temporal memory safety

Data races

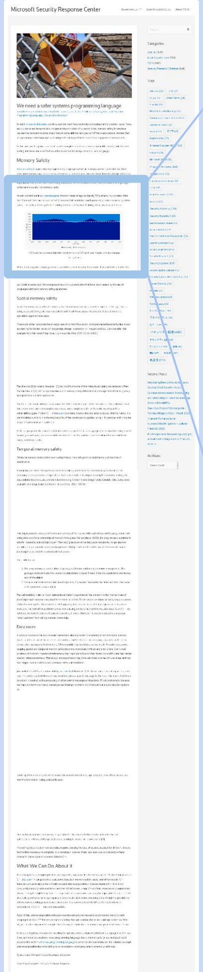
What We Can Do About It

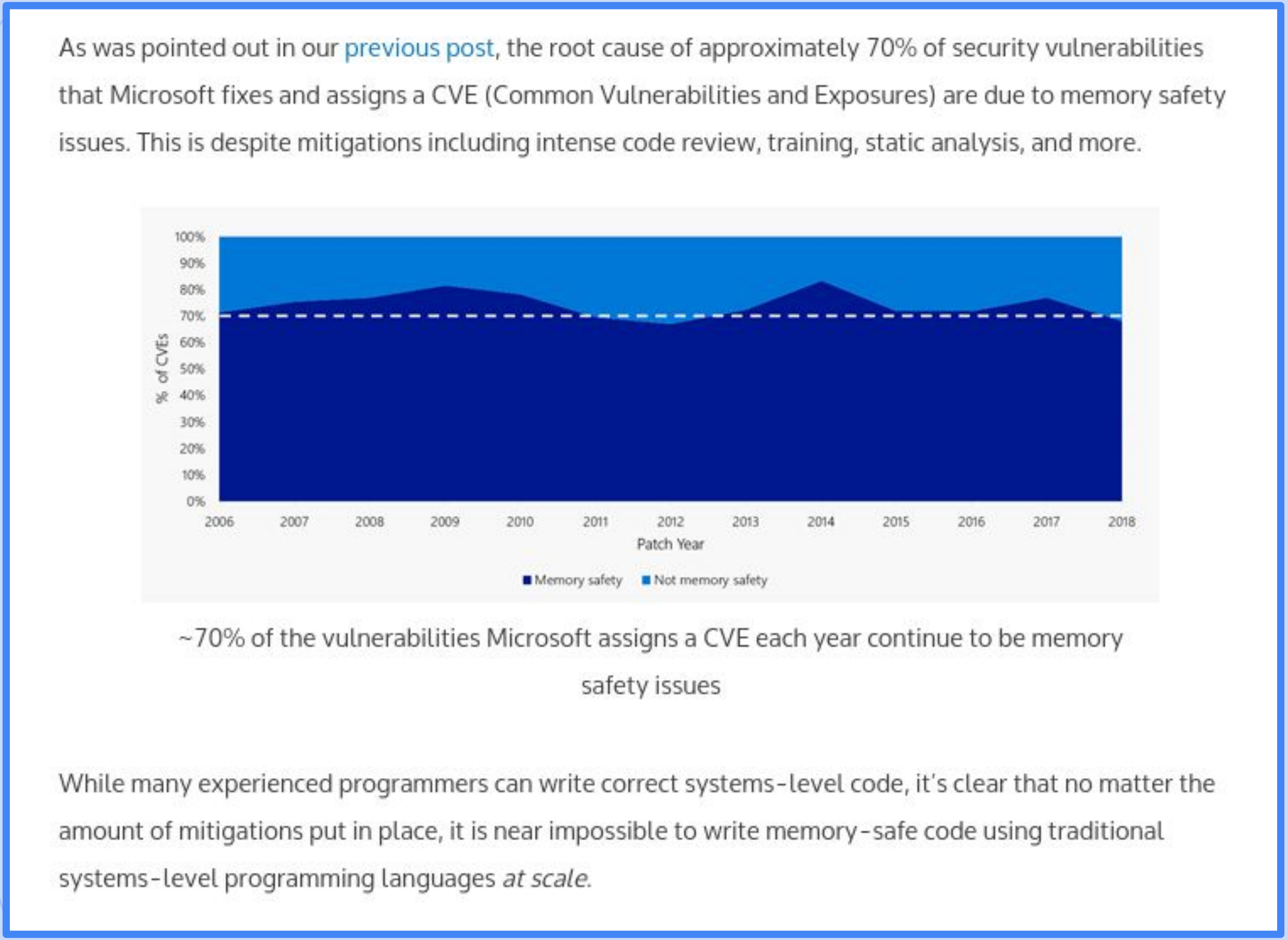
https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/



We need a safer systems programming language

Security Research & Defense / By MSRC Team / July 18, 2019 / Memory Safety, Rust, Safe Systems Programming Languages, Secure Development

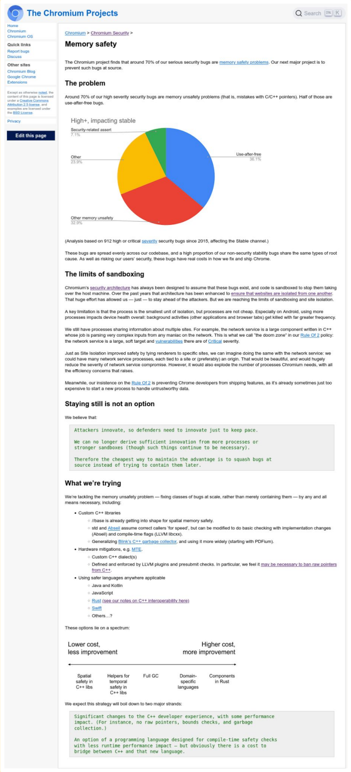

Microsoft Security Response Center
We need a safer systems programming language
Memory Safety



As was pointed out in our previous post, the root cause of approximately 70% of security vulnerabilities that Microsoft fixes and assigns a CVE (Common Vulnerabilities and Exposures) are due to memory safety issues. This is despite mitigations including intense code review, training, static analysis, and more.

~70% of the vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues

While many experienced programmers can write correct systems-level code, it's clear that no matter the amount of mitigations put in place, it is near impossible to write memory-safe code using traditional systems-level programming languages *at scale*.



Chromium > Chromium Security >

Memory safety

The Chromium project finds that around 70% of our serious security bugs are memory safety problems. Our next major project is to prevent such bugs at source.

The problem

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.

(Analysis based on 912 high or critical severity security bugs since 2015, affecting the Stable channel.)

These bugs are spread evenly across our codebase, and a high proportion of our non-security stability bugs share the same types of root cause. As well as risking our users' security, these bugs have real costs in how we fix and ship Chrome.

The limits of sandboxing

Chromium's security architecture has always been designed to assume that these bugs exist, and code is sandboxed to stop them taking over the host machine. Over the past years that architecture has been enhanced to ensure that websites are isolated from one another. That huge effort has allowed us — just — to stay ahead of the attackers. But we are reaching the limits of sandboxing and site isolation.

A key limitation is that the process is the smallest unit of isolation, but processes are not cheap. Especially on Android, using more processes impacts device health overall: background activities (other applications and browser tabs) get killed with far greater frequency.

We still have processes sharing information about multiple sites. For example, the network service is a large component written in C++ whose job is parsing very complex inputs from any maniac on the network. This is what we call "the doom zone" in our Rule Of 2 policy: the network service is a large, soft target and vulnerabilities there are of Critical severity.

Just as Site Isolation improved safety by tying renderers to specific sites, we can imagine doing the same with the network service: we could have many network service processes, each tied to a site or (preferably) an origin. That would be beautiful, and would hugely reduce the severity of network service compromise. However, it would also explode the number of processes Chromium needs, with all the efficiency concerns that raises.

Meanwhile, our insistence on the Rule Of 2 is preventing Chrome developers from shipping features, as it's already sometimes just too expensive to start a new process to handle untrustworthy data.

Staying still is not an option

We believe that:

```
Attackers innovate, so defenders need to innovate just to keep pace.

We can no longer derive sufficient innovation from more processes or
stronger sandboxes (though such things continue to be necessary).

Therefore the cheapest way to maintain the advantage is to squash bugs at
source instead of trying to contain them later.
```

What we're trying

We're tackling the memory unsafety problem — fixing classes of bugs at scale, rather than merely containing them — by any and all means necessary, including:

- Custom C++ libraries
  - //base is already getting into shape for spatial memory safety.
  - std and Abseil assume correct callers 'for speed', but can be modified to do basic checking with implementation changes (Abseil) and compile-time flags (LLVM libcxx).
  - Generalizing Blink's C++ garbage collector, and using it more widely (starting with PDFium).
- Hardware mitigations, e.g. MTE.
  - Custom C++ dialect(s)
  - Defined and enforced by LLVM plugins and presubmit checks. In particular, we feel it may be necessary to ban raw pointers from C++.
- Using safer languages anywhere applicable
  - Java and Kotlin
  - JavaScript
  - Rust (see our notes on C++ interoperability here)
  - Swift
  - Others…?

These options lie on a spectrum:

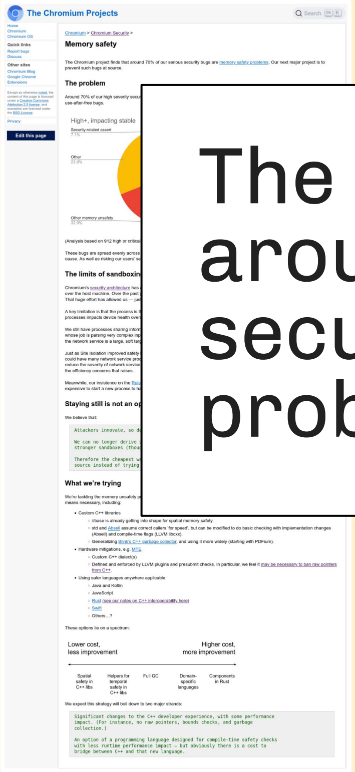
Lower cost,
less improvement
Higher cost,
more improvement
Spatial safety in C++ libs
Helpers for temporal safety in C++ libs
Full GC
Domain-specific languages
Components in Rust

We expect this strategy will boil down to two major strands:

```
Significant changes to the C++ developer experience, with some performance
impact. (For instance, no raw pointers, bounds checks, and garbage
collection.)

An option of a programming language designed for compile-time safety checks
with less runtime performance impact — but obviously there is a cost to
bridge between C++ and that new language.
```

https://www.chromium.org/Home/chromium-security/memory-safety/



Except as otherwise noted, the content of this page is licensed under a Creative Commons Attribution 2.5 license, and examples are licensed under the BSD License.

Chromium > Chromium Security >

Memory safety

The Chromium project finds that around 70% of our serious security bugs are memory safety problems. Our next major project is to prevent such bugs at source.

The problem

Around 70% of our high severity secu
use-after-free bugs.

(Analysis based on 912 high or critical

These bugs are spread evenly across
cause. As well as risking our users' se

The limits of sandboxin

Chromium's security architecture has
over the host machine. Over the past
That huge effort has allowed us — jus

A key limitation is that the process is t
processes impacts device health over

We still have processes sharing inform
whose job is parsing very complex inp
the network service is a large, soft targ

Just as Site Isolation improved safety
could have many network service proc
reduce the severity of network service
the efficiency concerns that raises.

Meanwhile, our insistence on the Rule
expensive to start a new process to ha

Staying still is not an op

We believe that:

```
Attackers innovate, so de

We can no longer derive s
stronger sandboxes (thoug

Therefore the cheapest wa
source instead of trying
```

What we're trying

We're tackling the memory unsafety p
means necessary, including:

- Custom C++ libraries
  - //base is already getting into shape for spatial memory safety.
  - std and Abseil assume correct callers 'for speed', but can be modified to do basic checking with implementation changes (Abseil) and compile-time flags (LLVM libcxx).
  - Generalizing Blink's C++ garbage collector, and using it more widely (starting with PDFium).
- Hardware mitigations, e.g. MTE.
  - Custom C++ dialect(s)
  - Defined and enforced by LLVM plugins and presubmit checks. In particular, we feel it may be necessary to ban raw pointers from C++.
- Using safer languages anywhere applicable
  - Java and Kotlin
  - JavaScript
  - Rust (see our notes on C++ interoperability here)
  - Swift
  - Others…?

These options lie on a spectrum:

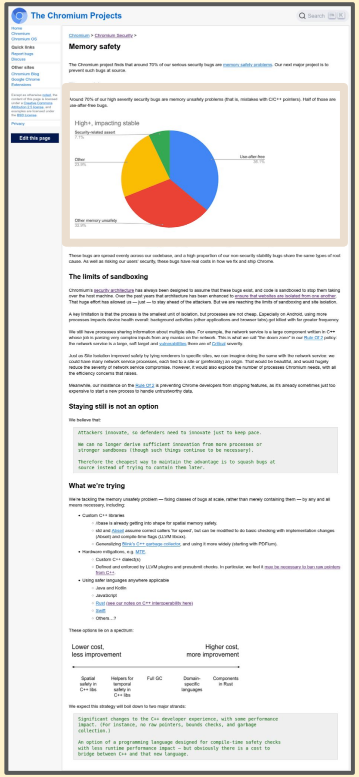
Lower cost,
less improvement
Higher cost,
more improvement
Spatial safety in C++ libs
Helpers for temporal safety in C++ libs
Full GC
Domain-specific languages
Components in Rust

We expect this strategy will boil down to two major strands:

```
Significant changes to the C++ developer experience, with some performance
impact. (For instance, no raw pointers, bounds checks, and garbage
collection.)

An option of a programming language designed for compile-time safety checks
with less runtime performance impact — but obviously there is a cost to
bridge between C++ and that new language.
```

The Chromium project finds that around 70% of our serious security bugs are memory safety problems.



The Chromium Projects

Search

Home
Chromium
Chromium OS

Quick links

Report bugs
Discuss

Other sites

Chromium Blog
Google Chrome
Extensions

Privacy

Edit this page

Chromium > Chromium Security >

Memory safety

The Chromium project finds that around 70% of our serious security bugs are memory safety problems. Our next major project is to prevent such bugs at source.

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.

These bugs are spread evenly across our codebase, and a high proportion of our non-security stability bugs share the same types of root cause. As well as risking our users' security, these bugs have real costs in how we fix and ship Chrome.

The limits of sandboxing

Chromium's security architecture has always been designed to assume that these bugs exist, and code is sandboxed to stop them taking over the host machine. Over the past years that architecture has been enhanced to ensure that websites are isolated from one another. That huge effort has allowed us — just — to stay ahead of the attackers. But we are reaching the limits of sandboxing and site isolation.

A key limitation is that the process is the smallest unit of isolation, but processes are not cheap. Especially on Android, using more processes impacts device health overall: background activities (other applications and browser tabs) get killed with far greater frequency.

We still have processes sharing information about multiple sites. For example, the network service is a large component written in C++ whose job is parsing very complex inputs from any maniac on the network. This is what we call "the doom zone" in our Rule Of 2 policy: the network service is a large, soft target and vulnerabilities there are of Critical severity.

Just as Site Isolation improved safety by tying renderers to specific sites, we can imagine doing the same with the network service: we could have many network service processes, each tied to a site or (preferably) an origin. That would be beautiful, and would hugely reduce the severity of network service compromise. However, it would also explode the number of processes Chromium needs, with all the efficiency concerns that raises.

Meanwhile, our insistence on the Rule Of 2 is preventing Chrome developers from shipping features, as it's already sometimes just too expensive to start a new process to handle untrustworthy data.

Staying still is not an option

We believe that:

```
Attackers innovate, so defenders need to innovate just to keep pace.

We can no longer derive sufficient innovation from more processes or
stronger sandboxes (though such things continue to be necessary).

Therefore the cheapest way to maintain the advantage is to squash bugs at
source instead of trying to contain them later.
```

What we're trying

We're tackling the memory unsafety problem — fixing classes of bugs at scale, rather than merely containing them — by any and all means necessary, including:

- Custom C++ libraries
    - //base is already getting into shape for spatial memory safety.
    - std and Abseil assume correct callers 'for speed', but can be modified to do basic checking with implementation changes (Abseil) and compile-time flags (LLVM libcxx).
    - Generalizing Blink's C++ garbage collector, and using it more widely (starting with PDFium).
- Hardware mitigations, e.g. MTE.
    - Custom C++ dialect(s)
    - Defined and enforced by LLVM plugins and presubmit checks. In particular, we feel it may be necessary to ban raw pointers from C++.
- Using safer languages anywhere applicable
    - Java and Kotlin
    - JavaScript
    - Rust (see our notes on C++ interoperability here)
    - Swift
    - Others…?

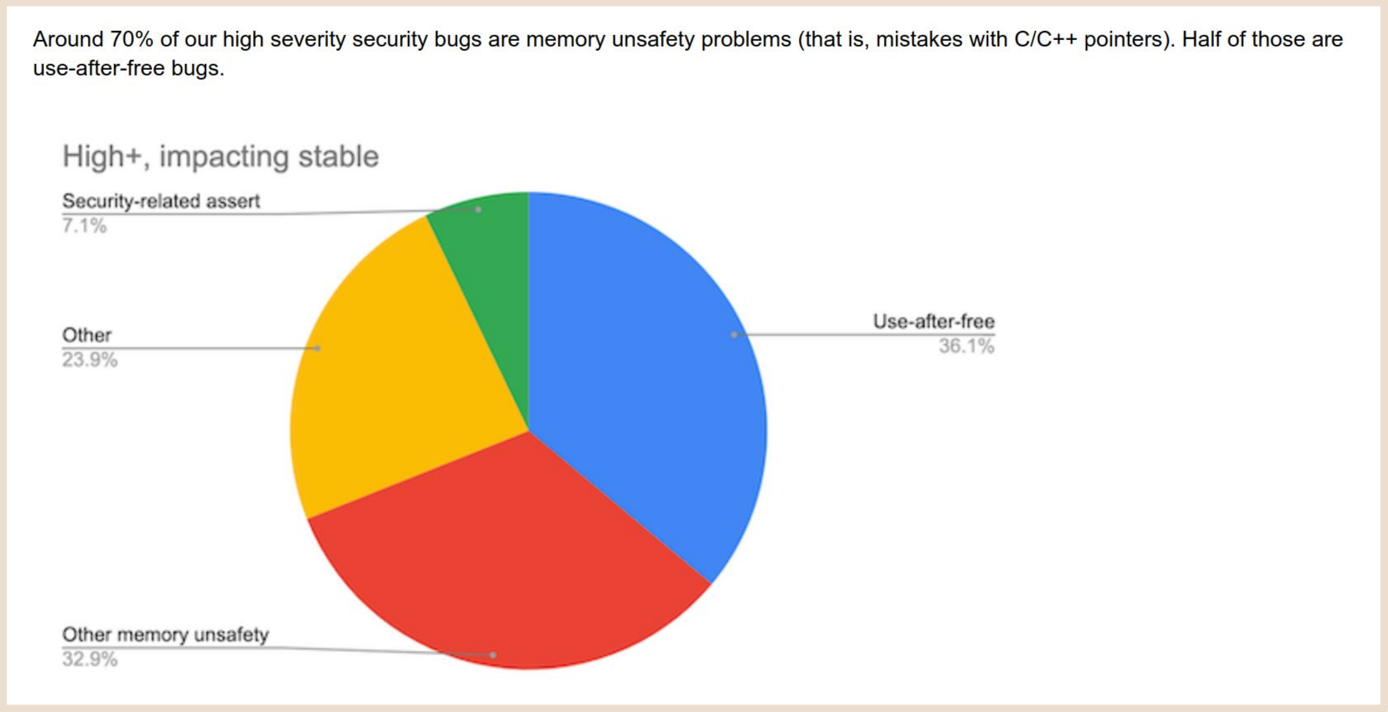
These options lie on a spectrum:

Lower cost,
less improvement
Higher cost,
more improvement
Spatial safety in C++ libs
Helpers for temporal safety in C++ libs
Full GC
Domain-specific languages
Components in Rust

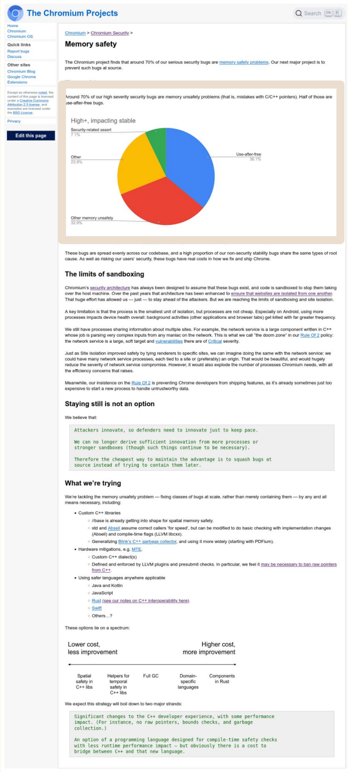
We expect this strategy will boil down to two major strands:

```
Significant changes to the C++ developer experience, with some performance
impact. (For instance, no raw pointers, bounds checks, and garbage
collection.)

An option of a programming language designed for compile-time safety checks
with less runtime performance impact — but obviously there is a cost to
bridge between C++ and that new language.
```



Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.



Memory safety

The Chromium project finds that around 70% of our serious security bugs are memory safety problems. Our next major project is to prevent such bugs at source.

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.

These bugs are spread evenly across our codebase, and a high proportion of our non-security stability bugs share the same types of root cause. As well as risking our users' security, these bugs have real costs in how we fix and ship Chrome.

The limits of sandboxing

Chromium's security architecture has always been designed to assume that these bugs exist, and code is sandboxed to stop them taking over the host machine. Over the past years that architecture has been enhanced to ensure that websites are isolated from one another. That huge effort has allowed us — just — to stay ahead of the attackers. But we are reaching the limits of sandboxing and site isolation.

A key limitation is that the process is the smallest unit of isolation, but processes are not cheap. Especially on Android, using more processes impacts device health overall: background activities (other applications and browser tabs) get killed with far greater frequency.

We still have processes sharing information about multiple sites. For example, the network service is a large component written in C++ whose job is parsing very complex inputs from any maniac on the network. This is what we call "the doom zone" in our Rule Of 2 policy: the network service is a large, soft target and vulnerabilities there are of Critical severity.

Just as Site Isolation improved safety by tying renderers to specific sites, we can imagine doing the same with the network service: we could have many network service processes, each tied to a site or (preferably) an origin. That would be beautiful, and would hugely reduce the severity of network service compromise. However, it would also explode the number of processes Chromium needs, with all the efficiency concerns that raises.

Meanwhile, our insistence on the Rule Of 2 is preventing Chrome developers from shipping features, as it's already sometimes just too expensive to start a new process to handle untrustworthy data.

Staying still is not an option

We believe that:

```
Attackers innovate, so defenders need to innovate just to keep pace.

We can no longer derive sufficient innovation from more processes or
stronger sandboxes (though such things continue to be necessary).

Therefore the cheapest way to maintain the advantage is to squash bugs at
source instead of trying to contain them later.
```

What we're trying

We're tackling the memory unsafety problem — fixing classes of bugs at scale, rather than merely containing them — by any and all means necessary, including:

- Custom C++ libraries
    - //base is already getting into shape for spatial memory safety.
    - std and Abseil assume correct callers 'for speed', but can be modified to do basic checking with implementation changes (Abseil) and compile-time flags (LLVM libcxx).
    - Generalizing Blink's C++ garbage collector, and using it more widely (starting with PDFium).
- Hardware mitigations, e.g. MTE.
    - Custom C++ dialect(s)
    - Defined and enforced by LLVM plugins and presubmit checks. In particular, we feel it may be necessary to ban raw pointers from C++.
- Using safer languages anywhere applicable
    - Java and Kotlin
    - JavaScript
    - Rust (see our notes on C++ interoperability here)
    - Swift
    - Others…?

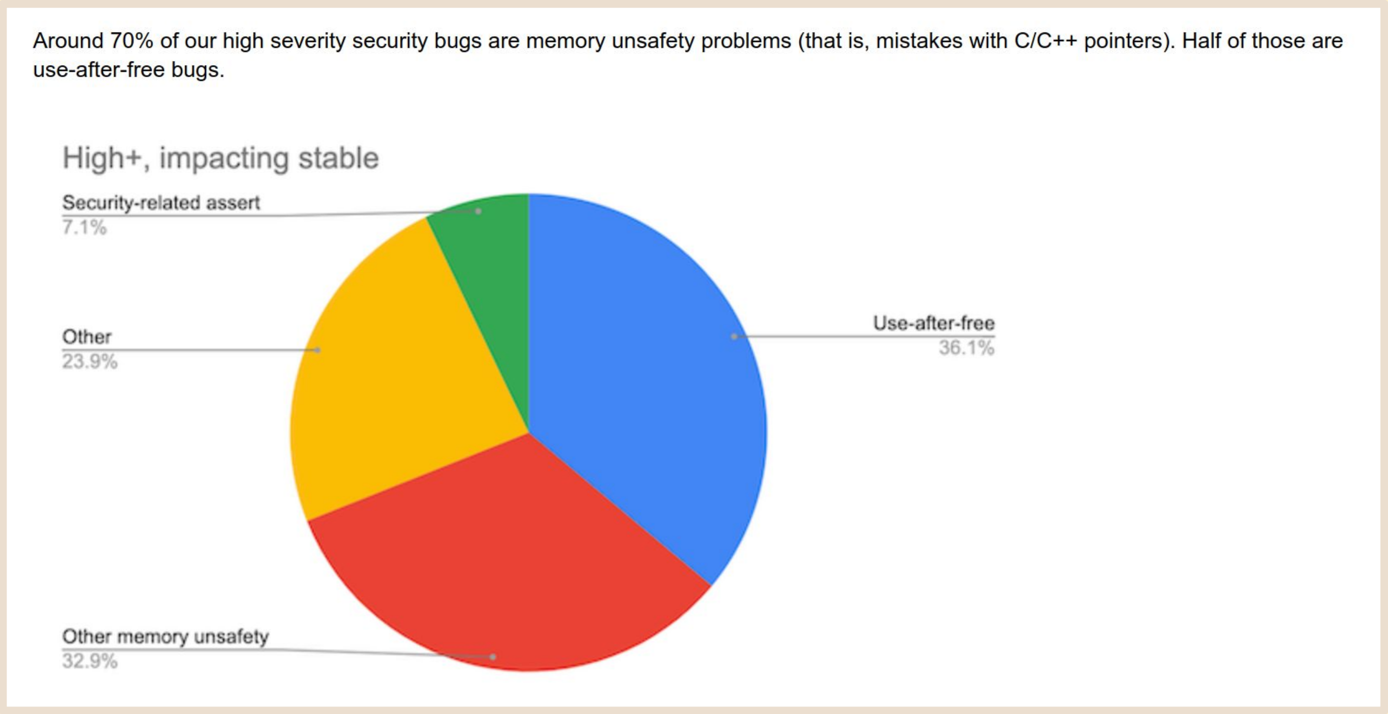
These options lie on a spectrum:

Lower cost,
less improvement
Higher cost,
more improvement
Spatial safety in C++ libs
Helpers for temporal safety in C++ libs
Full GC
Domain-specific languages
Components in Rust

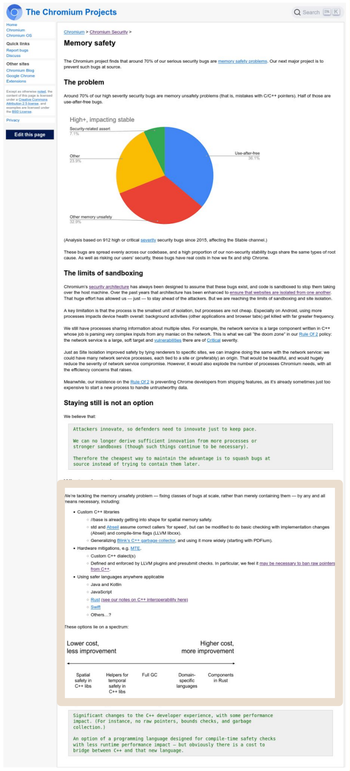
We expect this strategy will boil down to two major strands:

```
Significant changes to the C++ developer experience, with some performance
impact. (For instance, no raw pointers, bounds checks, and garbage
collection.)

An option of a programming language designed for compile-time safety checks
with less runtime performance impact — but obviously there is a cost to
bridge between C++ and that new language.
```



Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.



The Chromium Projects

Except as otherwise noted, the content of this page is licensed under a Creative Commons Attribution 2.5 license, and examples are licensed under the BSD License.

Chromium > Chromium Security >

Memory safety

The Chromium project finds that around 70% of our serious security bugs are memory safety problems. Our next major project is to prevent such bugs at source.

The problem

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.

(Analysis based on 912 high or critical severity security bugs since 2015, affecting the Stable channel.)

These bugs are spread evenly across our codebase, and a high proportion of our non-security stability bugs share the same types of root cause. As well as risking our users' security, these bugs have real costs in how we fix and ship Chrome.

The limits of sandboxing

Chromium's security architecture has always been designed to assume that these bugs exist, and code is sandboxed to stop them taking over the host machine. Over the past years that architecture has been enhanced to ensure that websites are isolated from one another. That huge effort has allowed us — just — to stay ahead of the attackers. But we are reaching the limits of sandboxing and site isolation.

A key limitation is that the process is the smallest unit of isolation, but processes are not cheap. Especially on Android, using more processes impacts device health overall: background activities (other applications and browser tabs) get killed with far greater frequency.

We still have processes sharing information about multiple sites. For example, the network service is a large component written in C++ whose job is parsing very complex inputs from any maniac on the network. This is what we call "the doom zone" in our Rule Of 2 policy: the network service is a large, soft target and vulnerabilities there are of Critical severity.

Just as Site Isolation improved safety by tying renderers to specific sites, we can imagine doing the same with the network service: we could have many network service processes, each tied to a site or (preferably) an origin. That would be beautiful, and would hugely reduce the severity of network service compromise. However, it would also explode the number of processes Chromium needs, with all the efficiency concerns that raises.

Meanwhile, our insistence on the Rule Of 2 is preventing Chrome developers from shipping features, as it's already sometimes just too expensive to start a new process to handle untrustworthy data.

Staying still is not an option

We believe that:

```
Attackers innovate, so defenders need to innovate just to keep pace.

We can no longer derive sufficient innovation from more processes or
stronger sandboxes (though such things continue to be necessary).

Therefore the cheapest way to maintain the advantage is to squash bugs at
source instead of trying to contain them later.
```

We're tackling the memory unsafety problem — fixing classes of bugs at scale, rather than merely containing them — by any and all means necessary, including:

- Custom C++ libraries
  - //base is already getting into shape for spatial memory safety.
  - std and Abseil assume correct callers 'for speed', but can be modified to do basic checking with implementation changes (Abseil) and compile-time flags (LLVM libcxx).
  - Generalizing Blink's C++ garbage collector, and using it more widely (starting with PDFium).
- Hardware mitigations, e.g. MTE.
  - Custom C++ dialect(s)
  - Defined and enforced by LLVM plugins and presubmit checks. In particular, we feel it may be necessary to ban raw pointers from C++.
- Using safer languages anywhere applicable
  - Java and Kotlin
  - JavaScript
  - Rust (see our notes on C++ interoperability here)
  - Swift
  - Others…?

These options lie on a spectrum:

Lower cost,
less improvement

Higher cost,
more improvement

Spatial safety in C++ libs

Helpers for temporal safety in C++ libs

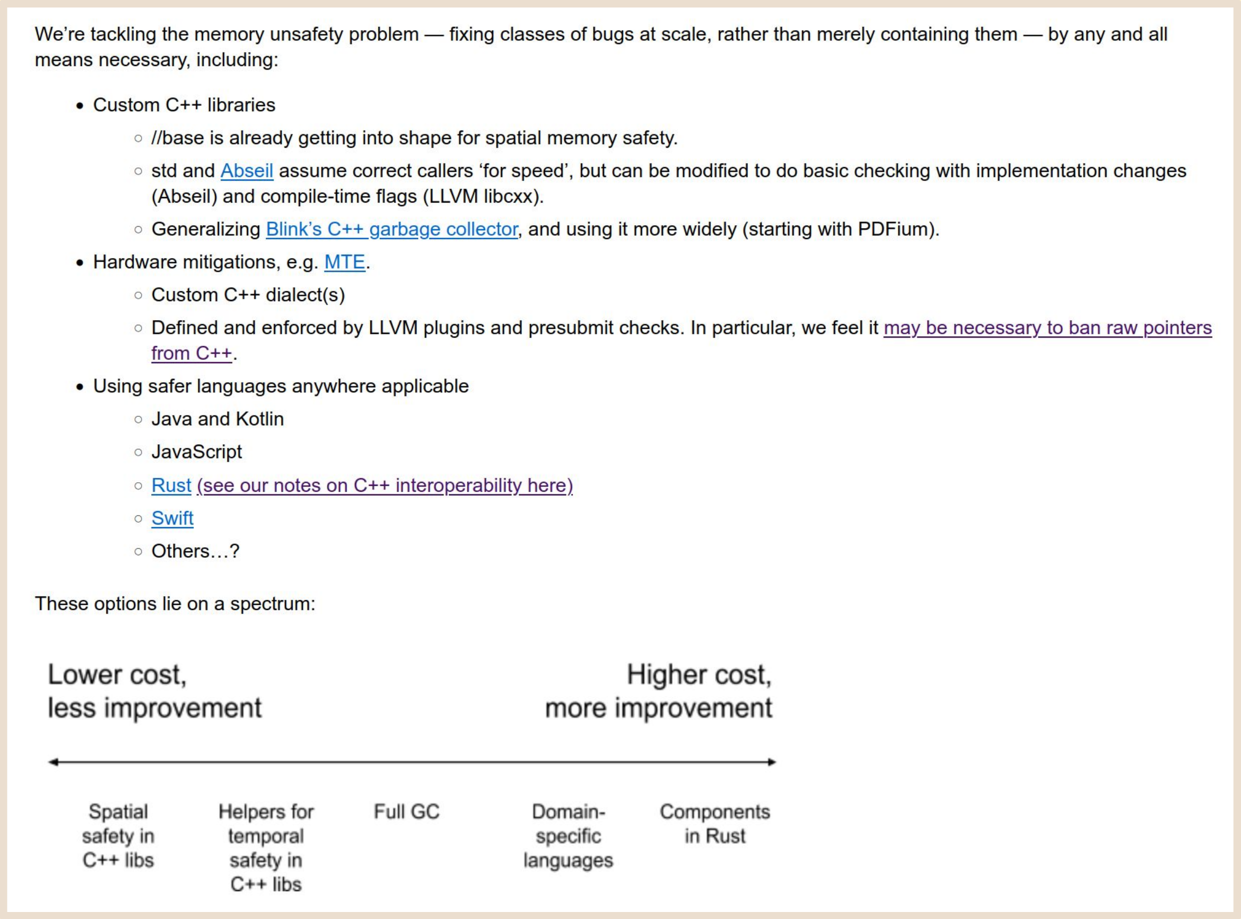
Full GC

Domain-specific languages

Components in Rust

```
Significant changes to the C++ developer experience, with some performance
impact. (For instance, no raw pointers, bounds checks, and garbage
collection.)

An option of a programming language designed for compile-time safety checks
with less runtime performance impact — but obviously there is a cost to
bridge between C++ and that new language.
```



We're tackling the memory unsafety problem — fixing classes of bugs at scale, rather than merely containing them — by any and all means necessary, including:

- Custom C++ libraries
  - //base is already getting into shape for spatial memory safety.
  - std and Abseil assume correct callers 'for speed', but can be modified to do basic checking with implementation changes (Abseil) and compile-time flags (LLVM libcxx).
  - Generalizing Blink's C++ garbage collector, and using it more widely (starting with PDFium).
- Hardware mitigations, e.g. MTE.
  - Custom C++ dialect(s)
  - Defined and enforced by LLVM plugins and presubmit checks. In particular, we feel it may be necessary to ban raw pointers from C++.
- Using safer languages anywhere applicable
  - Java and Kotlin
  - JavaScript
  - Rust (see our notes on C++ interoperability here)
  - Swift
  - Others…?

These options lie on a spectrum:

Lower cost,
less improvement

Higher cost,
more improvement

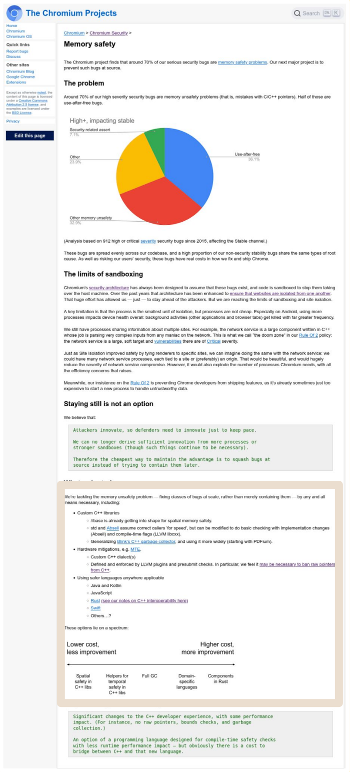
Spatial safety in C++ libs

Helpers for temporal safety in C++ libs

Full GC

Domain-specific languages

Components in Rust



Except as otherwise noted, the content of this page is licensed under a Creative Commons Attribution 2.5 license, and examples are licensed under the BSD License.

Chromium > Chromium Security >

Memory safety

The Chromium project finds that around 70% of our serious security bugs are memory safety problems. Our next major project is to prevent such bugs at source.

The problem

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.

(Analysis based on 912 high or critical severity security bugs since 2015, affecting the Stable channel.)

These bugs are spread evenly across our codebase, and a high proportion of our non-security stability bugs share the same types of root cause. As well as risking our users' security, these bugs have real costs in how we fix and ship Chrome.

The limits of sandboxing

Chromium's security architecture has always been designed to assume that these bugs exist, and code is sandboxed to stop them taking over the host machine. Over the past years that architecture has been enhanced to ensure that websites are isolated from one another. That huge effort has allowed us — just — to stay ahead of the attackers. But we are reaching the limits of sandboxing and site isolation.

A key limitation is that the process is the smallest unit of isolation, but processes are not cheap. Especially on Android, using more processes impacts device health overall: background activities (other applications and browser tabs) get killed with far greater frequency.

We still have processes sharing information about multiple sites. For example, the network service is a large component written in C++ whose job is parsing very complex inputs from any maniac on the network. This is what we call "the doom zone" in our Rule Of 2 policy: the network service is a large, soft target and vulnerabilities there are of Critical severity.

Just as Site Isolation improved safety by tying renderers to specific sites, we can imagine doing the same with the network service: we could have many network service processes, each tied to a site or (preferably) an origin. That would be beautiful, and would hugely reduce the severity of network service compromise. However, it would also explode the number of processes Chromium needs, with all the efficiency concerns that raises.

Meanwhile, our insistence on the Rule Of 2 is preventing Chrome developers from shipping features, as it's already sometimes just too expensive to start a new process to handle untrustworthy data.

Staying still is not an option

We believe that:

```
Attackers innovate, so defenders need to innovate just to keep pace.

We can no longer derive sufficient innovation from more processes or
stronger sandboxes (though such things continue to be necessary).

Therefore the cheapest way to maintain the advantage is to squash bugs at
source instead of trying to contain them later.
```

We're tackling the memory unsafety problem — fixing classes of bugs at scale, rather than merely containing them — by any and all means necessary, including:

- Custom C++ libraries
  - //base is already getting into shape for spatial memory safety.
  - std and Abseil assume correct callers 'for speed', but can be modified to do basic checking with implementation changes (Abseil) and compile-time flags (LLVM libcxx).
  - Generalizing Blink's C++ garbage collector, and using it more widely (starting with PDFium).
- Hardware mitigations, e.g. MTE.
  - Custom C++ dialect(s)
  - Defined and enforced by LLVM plugins and presubmit checks. In particular, we feel it may be necessary to ban raw pointers from C++.
- Using safer languages anywhere applicable
  - Java and Kotlin
  - JavaScript
  - Rust (see our notes on C++ interoperability here)
  - Swift
  - Others…?

These options lie on a spectrum:

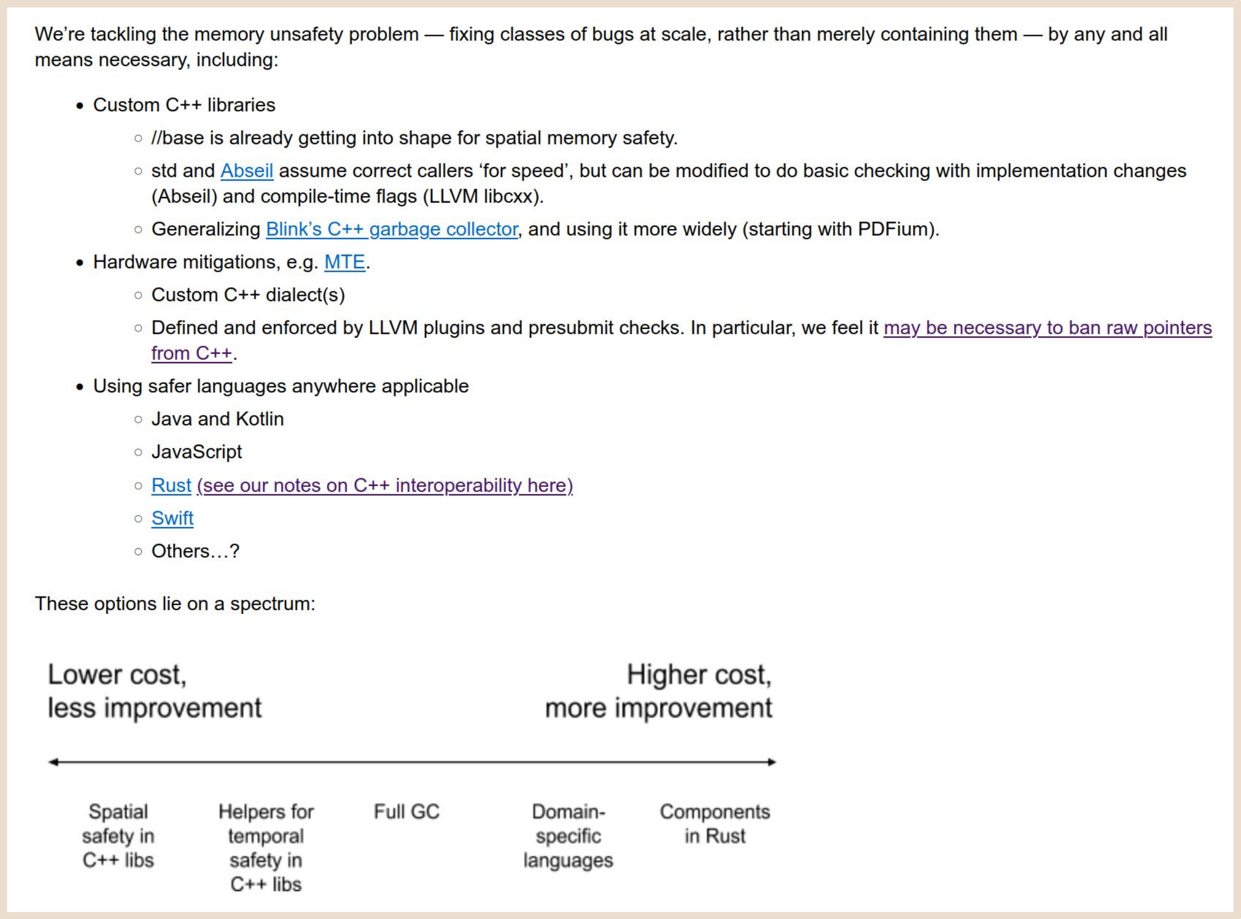
Lower cost,
less improvement

Higher cost,
more improvement

Spatial safety in C++ libs | Helpers for temporal safety in C++ libs | Full GC | Domain-specific languages | Components in Rust

```
Significant changes to the C++ developer experience, with some performance
impact. (For instance, no raw pointers, bounds checks, and garbage
collection.)

An option of a programming language designed for compile-time safety checks
with less runtime performance impact — but obviously there is a cost to
bridge between C++ and that new language.
```



We're tackling the memory unsafety problem — fixing classes of bugs at scale, rather than merely containing them — by any and all means necessary, including:

- Custom C++ libraries
  - //base is already getting into shape for spatial memory safety.
  - std and Abseil assume correct callers 'for speed', but can be modified to do basic checking with implementation changes (Abseil) and compile-time flags (LLVM libcxx).
  - Generalizing Blink's C++ garbage collector, and using it more widely (starting with PDFium).
- Hardware mitigations, e.g. MTE.
  - Custom C++ dialect(s)
  - Defined and enforced by LLVM plugins and presubmit checks. In particular, we feel it may be necessary to ban raw pointers from C++.
- Using safer languages anywhere applicable
  - Java and Kotlin
  - JavaScript
  - Rust (see our notes on C++ interoperability here)
  - Swift
  - Others…?

These options lie on a spectrum:

Lower cost,
less improvement

Higher cost,
more improvement

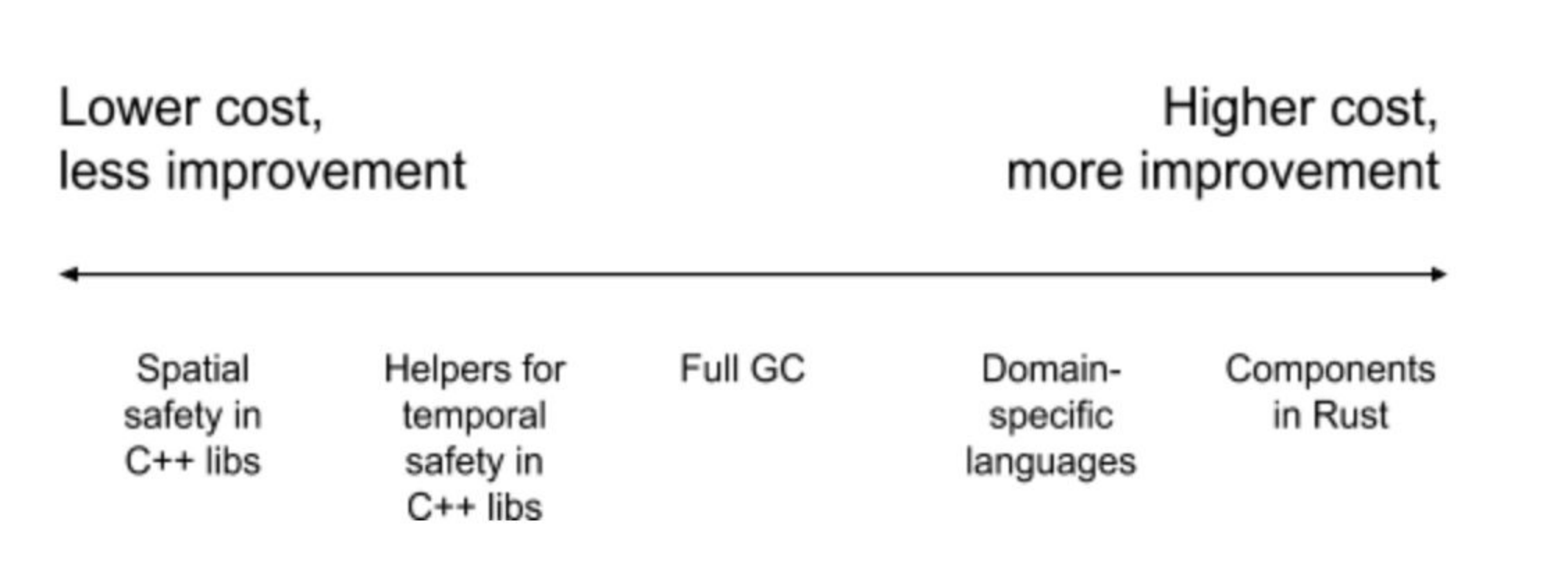
Spatial safety in C++ libs

Helpers for temporal safety in C++ libs

Full GC

Domain-specific languages

Components in Rust



Lower cost,
less improvement

Higher cost,
more improvement

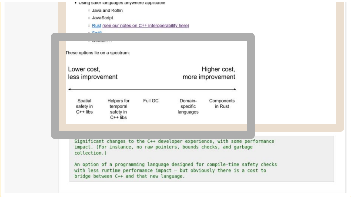
Spatial safety in C++ libs

Helpers for temporal safety in C++ libs

Full GC

Domain-specific languages

Components in Rust



- Using safer languages anywhere applicable
  - Java and Kotlin
  - JavaScript
  - Rust (see our notes on C++ interoperability here)
  - Swift
  - Others...?

These options lie on a spectrum:

Lower cost,
less improvement

Higher cost,
more improvement

Spatial safety in C++ libs

Helpers for temporal safety in C++ libs

Full GC

Domain-specific languages

Components in Rust

```
Significant changes to the C++ developer experience, with some performance
impact. (For instance, no raw pointers, bounds checks, and garbage
collection.)

An option of a programming language designed for compile-time safety checks
with less runtime performance impact — but obviously there is a cost to
bridge between C++ and that new language.
```

less improvement

more improvement

Spatial safety in C++ libs

Helpers for temporal safety in C++ libs

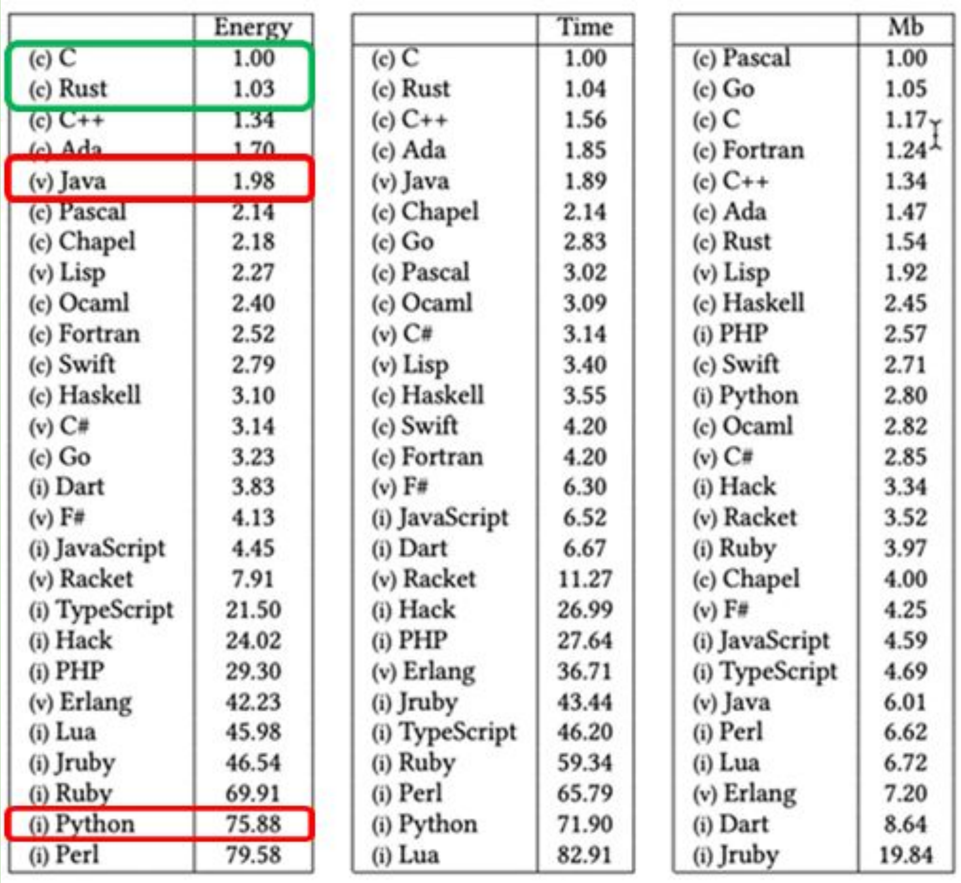
Full GC

Domain-specific languages

Components in Rust

Why Rust?

The planet is suffering



	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

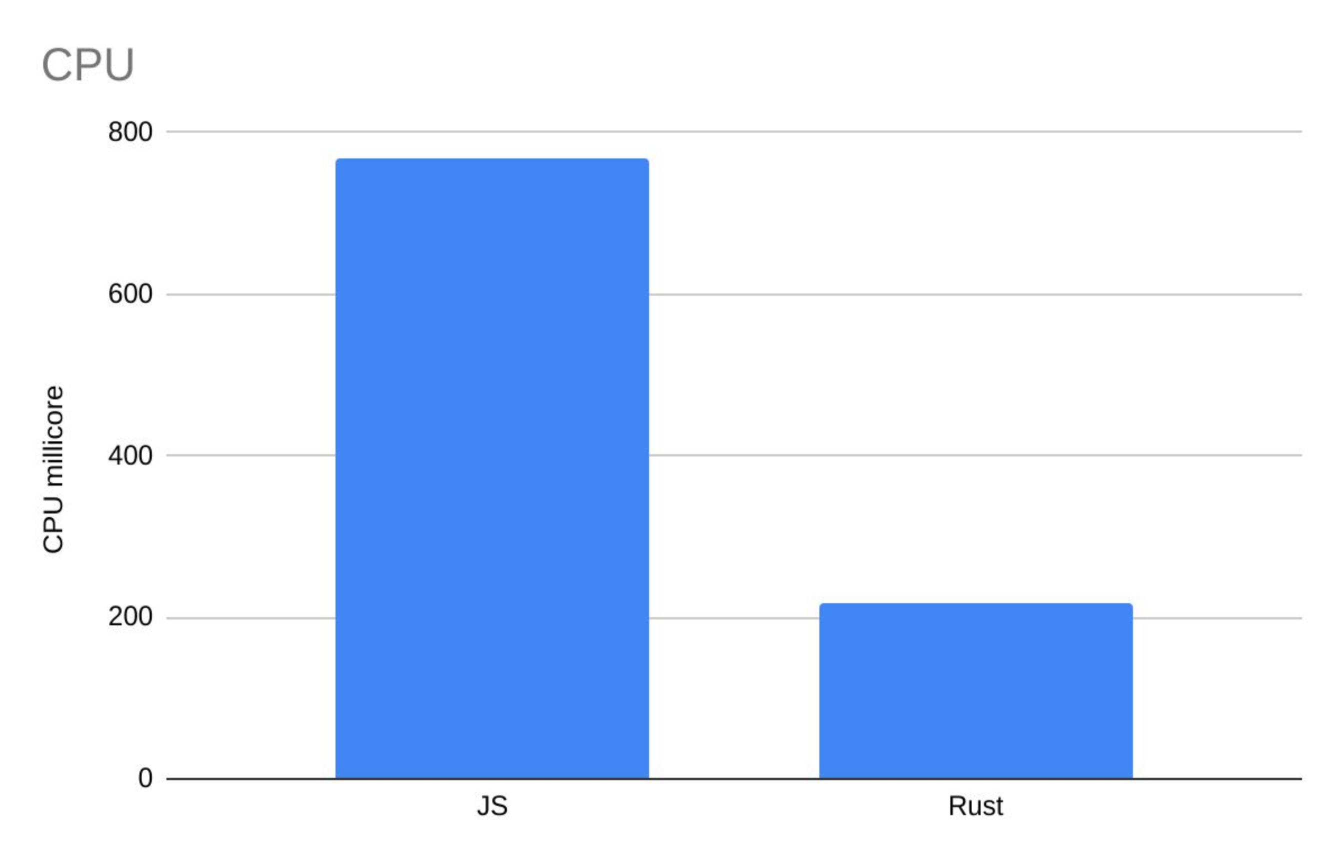
	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

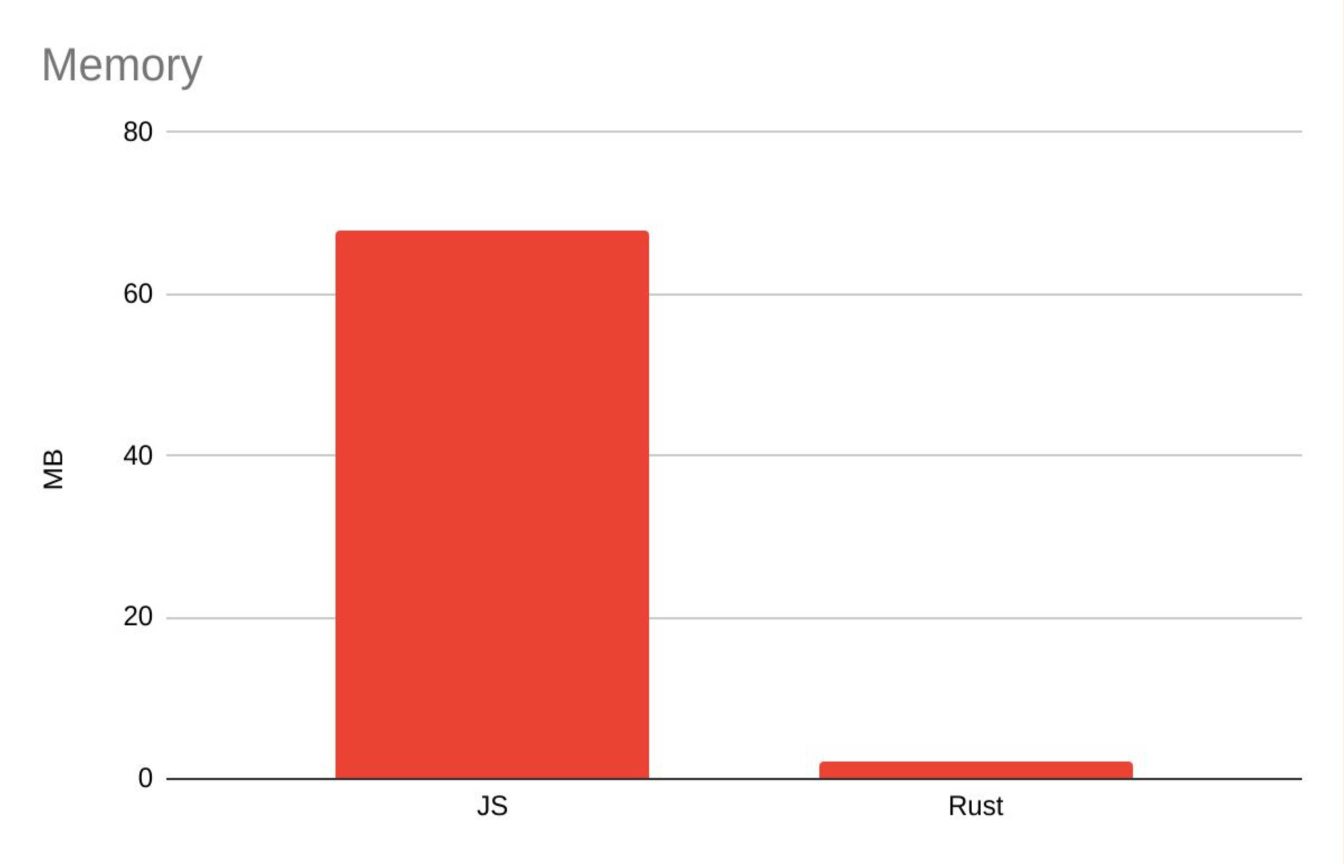
https://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf

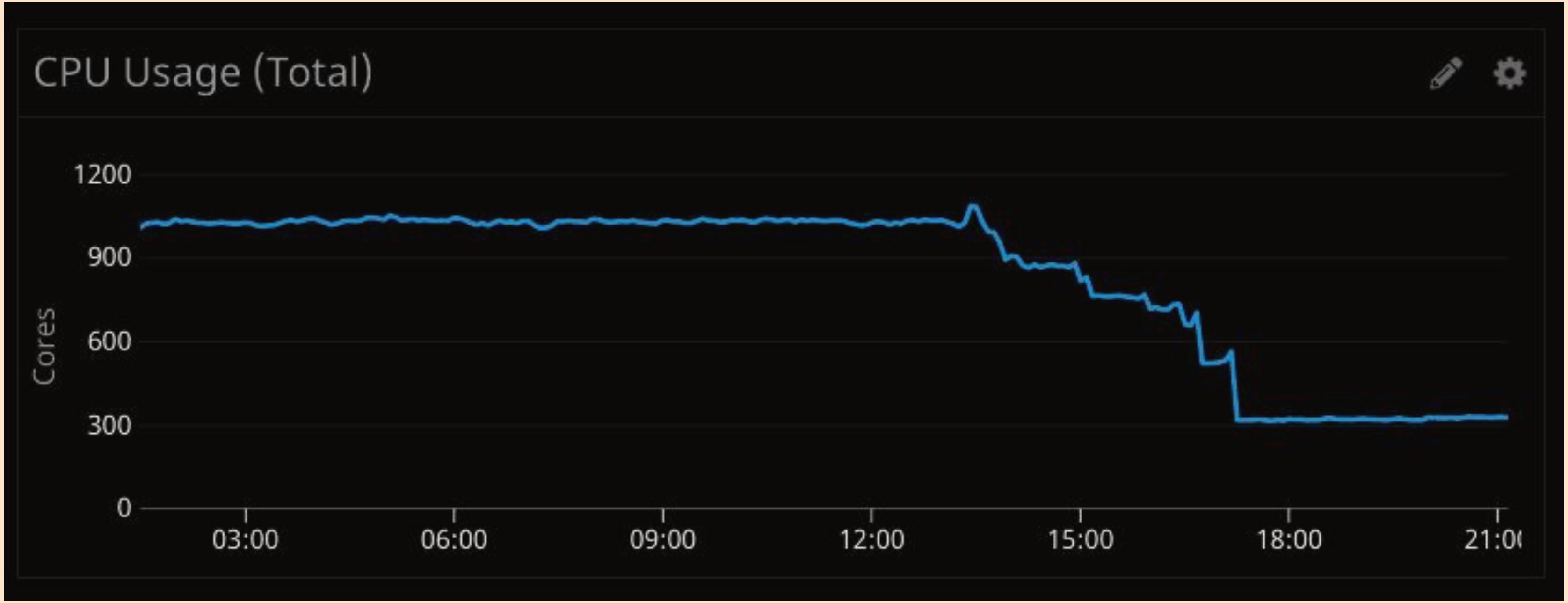
Why Rust?

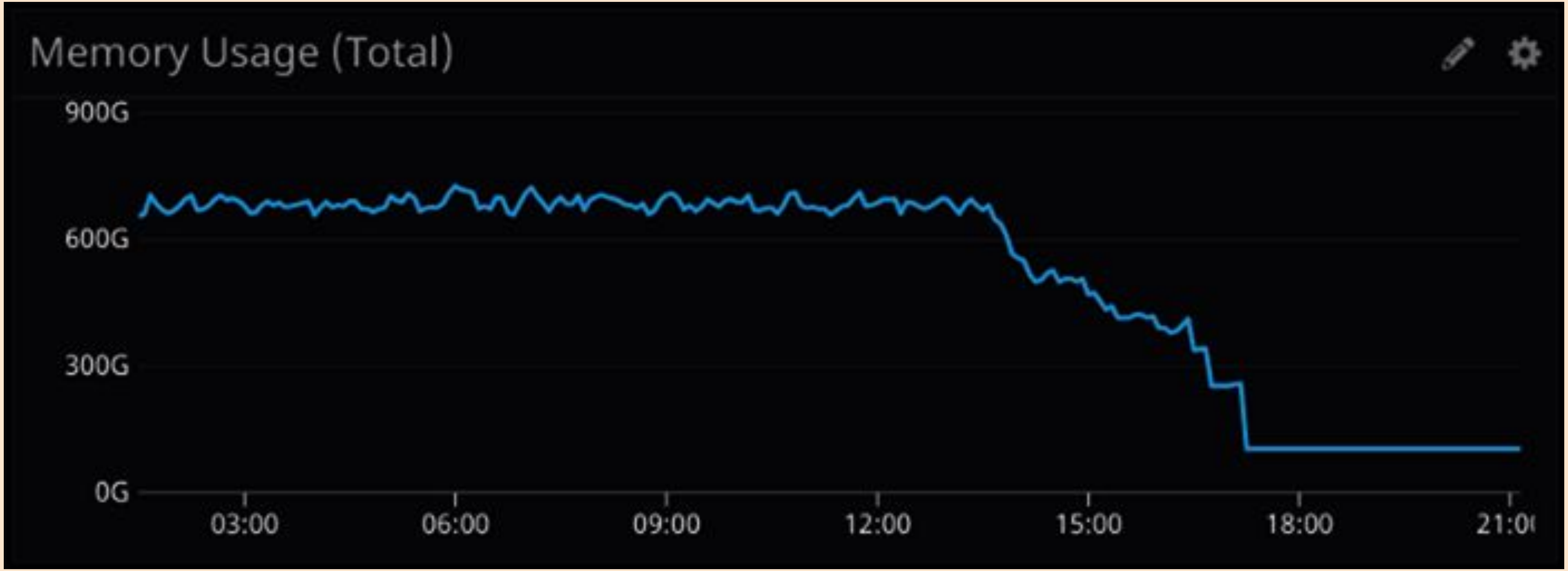
Save money



https://medium.com/tenable-techblog/optimizing-700-cpus-away-with-rust-dc7a000dbdb2

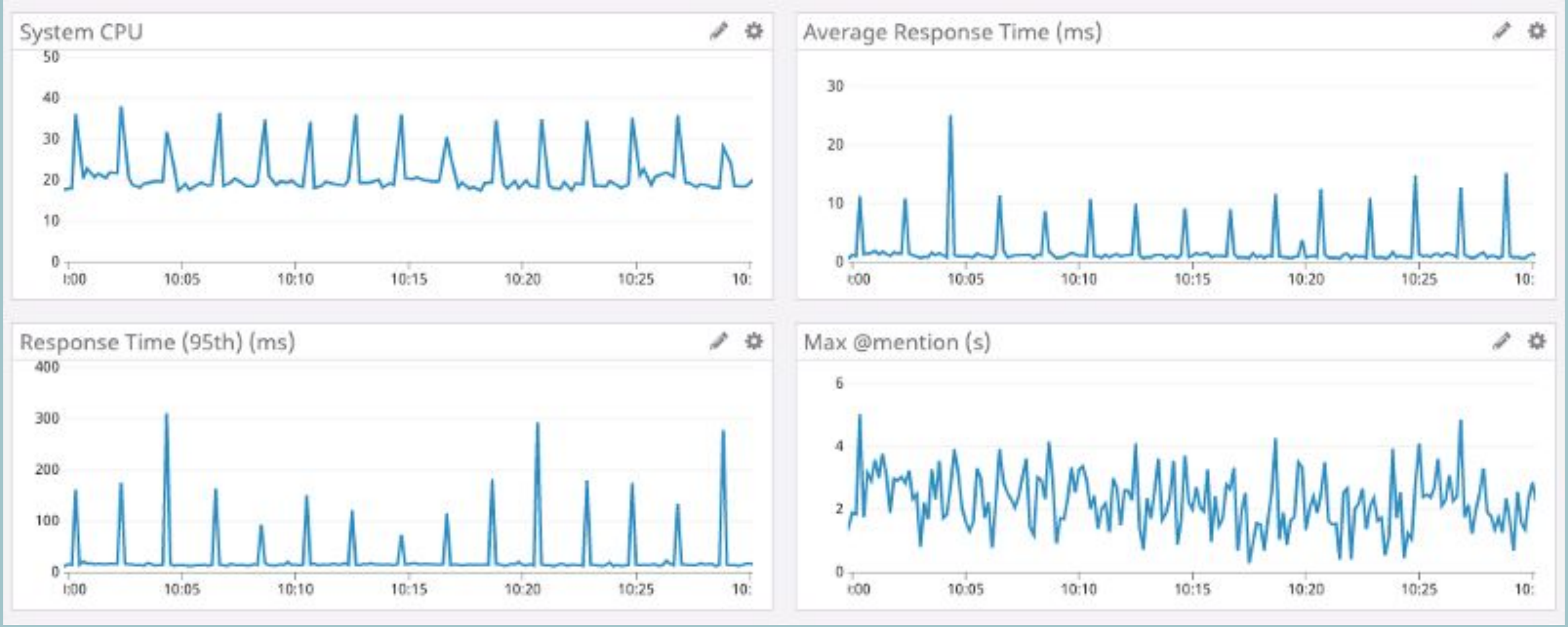

Memory
80
60
40
20
0
MB
JS
Rust


CPU Usage (Total)
Cores
1200
900
600
300
0
03:00
06:00
09:00
12:00
15:00
18:00
21:0

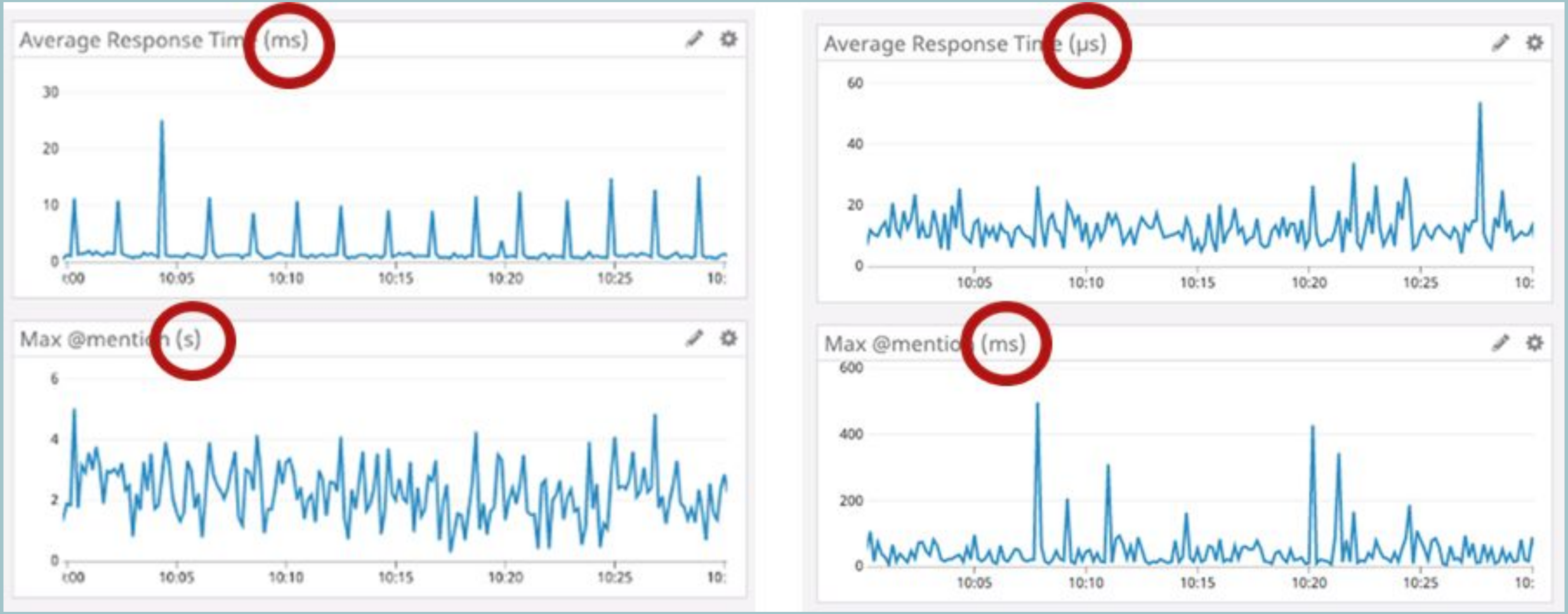

Memory Usage (Total)
900G
600G
300G
0G
03:00
06:00
09:00
12:00
15:00
18:00
21:0

Why Rust?

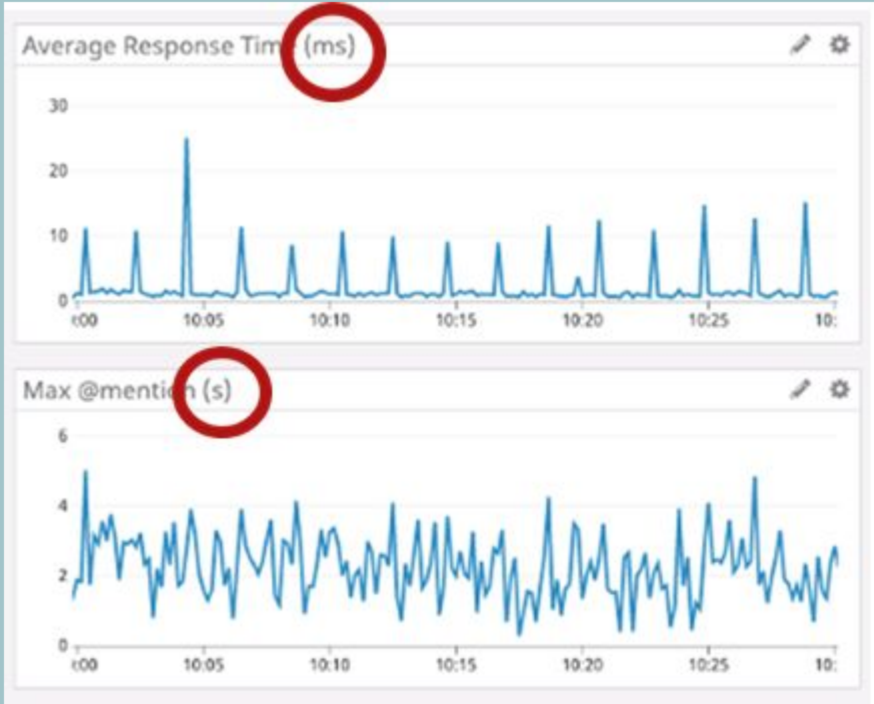
Increase stability



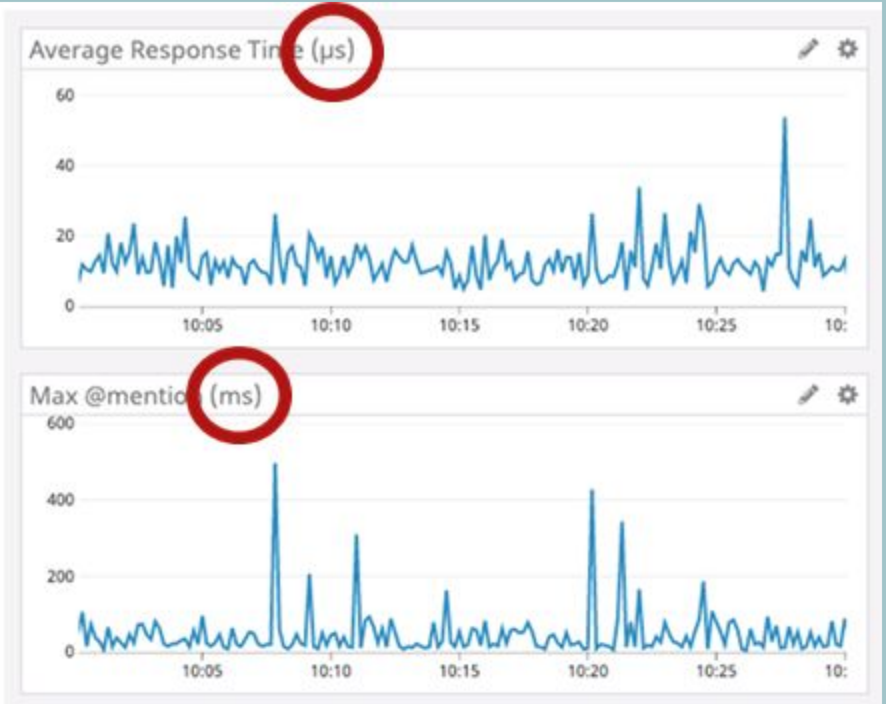
https://discord.com/blog/why-discord-is-switching-from-go-to-rust


Average Response Tim (ms)
30
20
10
0
:00
10:05
10:10
10:15
10:20
10:25
10:
Max @menti n (s)
6
4
2
0
:00
10:05
10:10
10:15
10:20
10:25
10:
Average Response Ti e (μs)
60
40
20
0
10:05
10:10
10:15
10:20
10:25
10:
Max @mentio (ms)
600
400
200
0
10:05
10:10
10:15
10:20
10:25
10:

Go

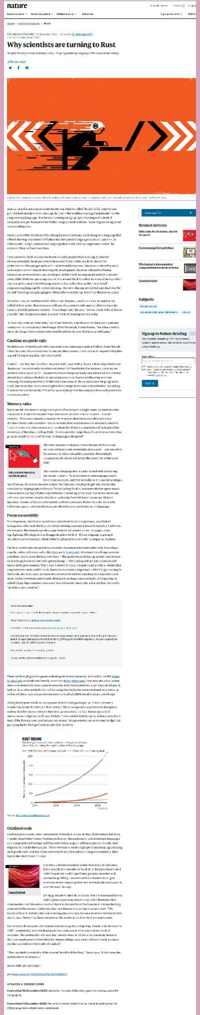

Average Response Tim (ms)
30
20
10
0
:00
10:05
10:10
10:15
10:20
10:25
10:
Max @menti n (s)
6
4
2
0
:00
10:05
10:10
10:15
10:20
10:25
10:

Rust


Average Response Ti e (μs)
60
40
20
0
10:05
10:10
10:15
10:20
10:25
10:
Max @mentio (ms)
600
400
200
0
10:05
10:10
10:15
10:20
10:25
10:

Why Rust?

Who wants bugs?



nature

TECHNOLOGY FEATURE

Why scientists are turning to Rust

Despite having a steep learning curve, the programming language offers speed and safety.

Jeffrey M. Perkel

Caution: no guide rails

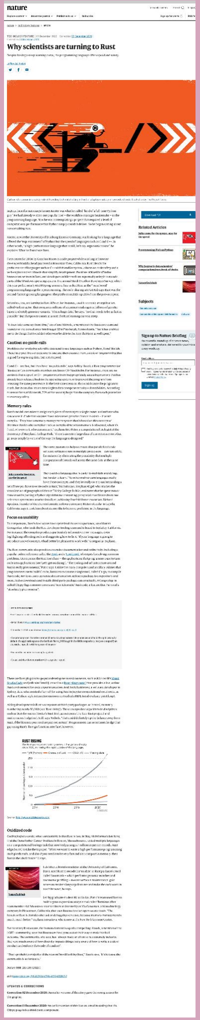
Memory rules

Focus on usability

RUST RISING

Oxidized code

Jeffrey M. Perkel. Nature 588, pp 185-186.
https://doi.org/10.1038/d41586-020-03382-2



nature

TECHNOLOGY FEATURE

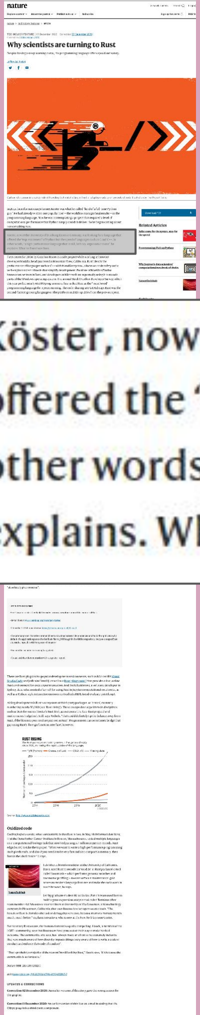
Why scientists are turning to Rust

Despite having a steep learning curve, the programming language offers speed and safety.

Jeffrey M. Perkel

Why scientists are turning to Rust

Jeffrey M. Perkel. Nature 588, pp 185-186.
https://doi.org/10.1038/d41586-020-03382-2



nature

TECHNOLOGY FEATURE

Why scientists are turning to Rust

Despite having a steep learning curve, the programming language offers speed and safety.

Jeffrey M. Perkel

Köster, now at the University of Duisburg-Essen in Germany, was looking for a language that offered the "expressiveness" of Python but the speed of languages such as C and C++. In other words, "a high-performance language that is still, let's say, ergonomic to use", he explains. What he found was Rust.

nature

Why scientists are turning to Rust

Despite having a steep learning curve, the programming language offers speed and safety.

Jeffrey M. Perkel

C and C++ are fast, but they have “no guide rails”, says Ashley Hauck, a Rust programmer (or ‘Rustacean’, as community members are known) in Stockholm. For instance, there are no controls that stop a C or C++ programmer from inappropriately accessing memory that has already been released back to the operating system, or to prevent the program from releasing the same piece twice. In the best-case scenario, this would cause the program to crash. But it can also return meaningless data or expose security vulnerabilities. According to researchers at Microsoft, 70% of the security bugs that the company fixes each year relate to memory safety.

nature

Why scientists are turning to Rust

Despite having a steep learning curve, the programming language offers speed and safety.

Rust's model uses rules to assign each piece of memory to a single owner and enforce who can access it. Code that violates those rules never gets the chance to crash – it won't compile. "They have a memory-management system that is based on this concept of lifetimes that lets the compiler track at compile-time when memory is allocated, when it's freed, who owns it, who can access it," explains Rob Patro, a computational biologist at the University of Maryland, College Park. "There's an entire large class of correctness errors that go away simply by virtue of the way the language is designed."

nature

Why scientists are turning to Rust

But for many Rustaceans, the human element is equally compelling. Hauck, a member of the LGBT+ community, says that Rust users have gone out of their way to make her feel welcome. The community, she says, has “always made an effort to be extremely inclusive — like, very much aware of how diversity impacts things; very aware of how to write a code of conduct and enforce that code of conduct”.

“That’s probably a majority of the reason I’m still writing Rust,” Hauck says. “It’s because the community is so fantastic.”

Why Rust?

Your team will be happier

Rust is developers'
most loved programming language and
most preferred programming language

https://survey.stackoverflow.co/2022/

“The short answer is that Rust solves pain points present in many other languages, providing a solid step forward with a limited number of downsides.”

https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/

> “I believe that Rust is challenging to learn but rewarding to use. I think it is actually surprising how much people enjoy being challenged as long as the reward is good enough.”

https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/

> “When you’re outside of Rust, there are things that sound like empty slogans, but when you start using it you’ll become pleasantly surprised …”

https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/

Should you choose Rust?

Should you choose Rust?

- Can you afford to lower productivity in short term?
- Do you have an area of your business that could benefit?

Should you choose Rust?

If you're unsure, test

- The 2021 user survey reveals that Rust did not justify its adoption approximately 1% of the time
- https://raw.githubusercontent.com/rust-lang/surveys/main/surveys/2021-annual-survey/2021-annual-survey-summary.pdf, p 59

How to learn Rust

Preparation

- Give yourself permission to be frustrated

- Expect programs to be rejected that you feel should be accepted

How to learn Rust

Three steps

- Write small scripts
- Reimplement small service
- Implement larger project

Your aim

- Understand how Rust provides its guarantees and apply them across your business

How to adopt Rust

How to adopt Rust

Preparation

- Find low risk projects

How to adopt Rust

Three steps

- Find local advocate
- Reimplement small service
- Implement larger project

Your aim

- Understand how Rust provides its guarantees and apply them across your business

Where is Rust weak?

Where is Rust weak?

Learning

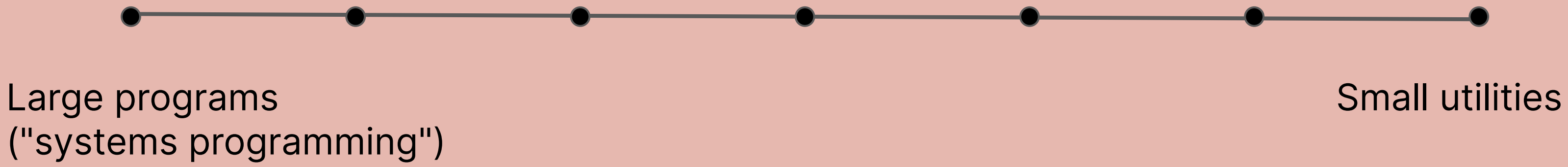
- 55% of people who have left the language community found Rust too hard

Which projects are a good fit?

Which projects are a good fit?

Large programs
("systems programming")

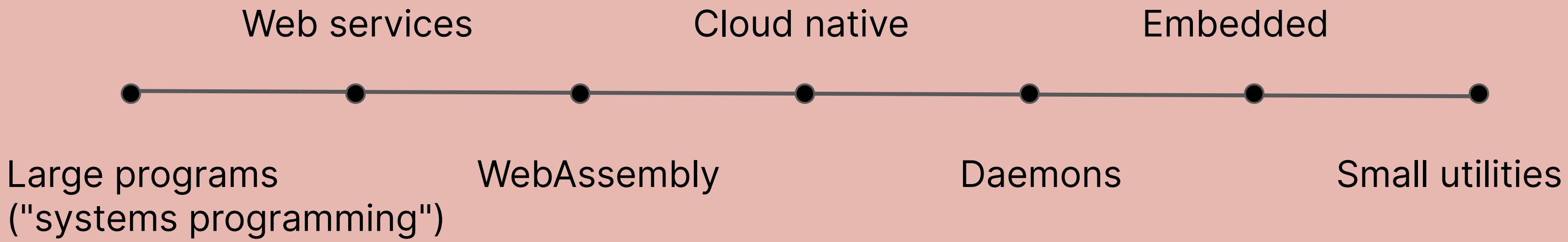
Which projects are a good fit?



Large programs
("systems programming")

Small utilities

Which projects are a good fit?



Large programs
("systems programming")

Web services

WebAssembly

Cloud native

Daemons

Embedded

Small utilities

You should consider Rust