Safety e Security del Codice

Incidenti e Soluzioni

Francesco Bernini
Università di Parma

ABSTRACT

La sicurezza e l'affidabilità del software sono questioni centrali nello sviluppo di sistemi informatici, specialmente in contesti *safety-critical* come l'aerospaziale, l'automotive e il biomedicale. La maggior parte delle vulnerabilità software derivano da errori nella gestione della memoria e della concorrenza, con conseguenze potenzialmente catastrofiche. Questo lavoro esplora le differenze tra **safety** e **security** del codice, evidenziando i principali problemi legati alla programmazione in linguaggi tradizionali come `C` e `C++`. Vengono analizzati casi storici di fallimenti software, come il Morris Worm, il disastro del razzo Ariane 5 e gli incidenti legati al Therac-25, per illustrare le implicazioni di un codice insicuro. Successivamente, si esaminano le soluzioni adottate nel tempo, tra cui tecniche di analisi statica, definizione di sottoinsiemi sicuri di linguaggi esistenti (come `MISRA C` e `Safe C++`) e l'approccio innovativo del linguaggio `Rust`, che offre meccanismi integrati per la gestione sicura della memoria senza il costo di un *garbage collector*. L'obiettivo di questa ricerca è quello di evidenziare l'importanza del codice sicuro e di informare su alcune delle tecniche di mitigazione del rischio esistenti allo stato dell'arte.

1. INTRODUZIONE

Negli ultimi anni, la sicurezza informatica è diventata una priorità per aziende, governi e sviluppatori, complice la crescente sofisticatezza degli attacchi e delle vulnerabilità software. La maggior parte di questi problemi origina da una erronea gestione della memoria [15], ma anche un'imprecisa gestione della concorrenza, un errata configurazione di sistemi o errori logici di programmazione possono portare a malfunzionamenti.

Il software è ormai parte integrante di quasi tutte le attività umane ed alcune di esse possono descriversi come "safety-critical", ovvero dove la sicurezza del corretto funzionamento del codice (anche in caso di situazioni eccezionali) deve essere garantita. Tra queste attività troviamo sicuramente tutte quelle che coinvolgono l'ambito avionico, automotive, biomedicale perché possono avere ripercussioni dirette sulla salute. Per lo sviluppo di questa tipologia di programmi vi sono linguaggi, tecniche di programmazione e software di supporto ad-hoc.

Ovviamente non tutti i tipi di software hanno questi requisiti stringenti di correttezza, come quelli relativi al mondo videoludico o al mondo della Data Science, per citarne alcuni. Tuttavia errori in entrambe queste categorie possono portare ad ingenti perdite di denaro, per motivi diversi [6] [9].

In generale non è sempre possibile scegliere un linguaggio di programmazione che offre funzionalità volte alla gestione della memoria automatica come il *garbage collector* perché esso richiede risorse computazionali che in certi ambiti non sono disponibili e inoltre rende impossibile determinare il tempo entro il quale certe operazioni verranno effettuate (requisito fondamentale dei sistemi real-time). Sono comunque disponibili diverse soluzioni che verranno trattate in questo documento.

2. SAFETY E SECURITY DEL CODICE

Nel contesto dello sviluppo software, safety e security sono due concetti strettamente correlati ma distinti.

La **safety** si riferisce alla capacità di un sistema di evitare guasti che possano portare a conseguenze pericolose, garantendo che il software si comporti in modo prevedibile anche in condizioni anomale. Nei linguaggi di programmazione, la safety si traduce nell'adozione di meccanismi che riducono la probabilità di errori critici, come la *type safety* (controllo rigoroso dei tipi di dato), l'assenza di *undefined behavior* (`UB`) e l'uso di strumenti per la verifica formale del codice. Linguaggi come `Ada` e `Rust` offrono forti garanzie intrinseche in questo senso, con controlli statici avanzati e sistemi di gestione della memoria sicuri. Per quanto riguarda i linguaggi maggiormente utilizzati attualmente negli ambiti dove è richiesta anche un minimo di performance oltre alla safety, troviamo linguaggi come `C` e `C++` che non impongono il controllo rigoroso dei tipi e permettono operazioni non definite dal linguaggio (`UB`) aumentando il rischio di comportamenti imprevedibili. Tuttavia, essi offrono tutti i costrutti per applicare la safety, ma la responsabilità è affidata al programmatore.

La **security**, invece, riguarda la protezione del software da attacchi malevoli e accessi non autorizzati. Un sistema è *secure*

se resiste a tentativi di compromissione, proteggendo dati e funzionalità da utenti non autorizzati o codice dannoso. Nei linguaggi di programmazione, la security è anch'essa influenzata dalla gestione della memoria (assenza di vulnerabilità come *buffer overflow* o *use-after-free*), dai meccanismi di isolamento tra processi (*multithreading*) e, inoltre, dalla disponibilità di strumenti per la gestione sicura delle credenziali e della crittografia. Linguaggi moderni come `Rust` e `Go` sono progettati con un forte focus sulla sicurezza, prevenendo classi di vulnerabilità comuni.

Sebbene safety e security abbiano obiettivi diversi, esistono molte sovrapposizioni. Ad esempio, una vulnerabilità di sicurezza può compromettere la safety di un sistema: un attacco informatico ad un software aeronautico scritto in `C`, che sfrutti un buffer overflow, potrebbe manipolare i dati di controllo con conseguenze catastrofiche. Per questo motivo, i linguaggi di programmazione utilizzati in contesti critici devono essere progettati per minimizzare sia i rischi legati alla safety che quelli legati alla security, attraverso controlli statici, verifiche formali e restrizioni sulle operazioni potenzialmente pericolose.

3. INCIDENTI

In questa sezione verranno presentati alcuni dei più famosi eventi catastrofici causati da vulnerabilità dovute ad errori di programmazione.

3.1 Morris Worm

Il **Morris Worm**, rilasciato nel 1988, è considerato il primo *worm* informatico su Internet. Inizialmente concepito con l'idea di dimostrare l'esistenza di vulnerabilità facilmente sfruttabili nei sistemi Unix, il programma venne lanciato con le capacità di replicarsi su altre macchine attraverso la rete e di autodistruggersi allo spegnimento delle stesse. Tuttavia, all'epoca, lo spegnimento dei computer avveniva raramente e inoltre il comportamento programmato prevedeva la possibilità che una stessa macchina venisse infettata più volte. Questi fattori portarono il programma ad avere un effetto devastante: infettò circa 6.000 computer (circa il 10% dell'allora Internet) causando rallentamenti e crash di sistemi causando praticamente uno tra i primi "attacchi" DoS (Denial of Service).

Un worm è un tipo di malware che, a differenza di un virus che necessita dell'interazione dell'utente per essere eseguito, si replica e diffonde autonomamente da un computer all'altro sfruttando vulnerabilità di sicurezza come falle nei protocolli di rete o nei software di sistema senza bisogno di intervento umano.

Questo, in particolare, sfruttava una vulnerabilità di buffer overflow presente nel demone `finger`, un programma utilizzato per recuperare informazioni sugli utenti di un sistema Unix. Il codice, scritto in `C`, non eseguiva controlli adeguati sulla dimensione dei dati ricevuti, permettendo all'attaccante di scrivere dati oltre i limiti del buffer e sovrascrivere la memoria adiacente, potenzialmente eseguendo codice arbitrario. Il worm sfruttava anche altre falle di sicurezza, come una debolezza nel comando `sendmail`, che consentiva l'esecuzione remota di codice non autorizzato.

Il Morris Worm portò alla creazione del CERT (Computer Emergency Response Team) e alla consapevolezza della necessità di migliori pratiche di sicurezza nella programmazione. Infatti, stimolò lo sviluppo di protezioni contro attacchi come ASLR (Address Space Layout Randomization), che rende più difficile prevedere la posizione della memoria utilizzata da un programma, e Stack Canaries, una tecnica che protegge la memoria dello stack da sovrascritture accidentali o malevole.

3.2 Ariane 5

Il fallimento del volo inaugurale del razzo Ariane 5, avvenuto il 4 giugno 1996, è uno degli esempi più noti di disastro informatico causato da un errore software. Dopo soli 37 secondi dal lancio, il razzo si disintegrò in volo, provocando una perdita economica di circa 370 milioni di dollari.

La causa del fallimento fu un errore nella gestione della conversione di un numero in virgola mobile in un numero intero a 16 bit all'interno del sistema di guida inerziale del razzo, nonostante fosse scritto in un linguaggio fortemente tipizzato come `Ada`. Il codice riutilizzava parti di quello impiegato per Ariane 4, dove le condizioni di volo erano diverse e l'errore non si verificava. Tuttavia, il nuovo razzo Ariane 5 aveva una velocità significativamente maggiore durante la fase iniziale del volo, il che portò alla generazione di un valore superiore a quello rappresentabile dal tipo di dato utilizzato. Il tentativo di conversione generò un overflow, causando il crash del sistema di guida [10].

L'incidente di Ariane 5 evidenzia diversi problemi legati alla sicurezza software nei sistemi critici:

1. **Riutilizzo del codice senza adeguata validazione**: il software era stato progettato per Ariane 4 e non era stato testato adeguatamente per le nuove condizioni di Ariane 5.
2. **Gestione degli errori inadeguata**: una segnalazione di errore di un sottosistema critico non correttamente gestita portò alla perdita del razzo.
3. **Problemi di conversione dei tipi di dati**: errori nella conversione numerica possono avere conseguenze catastrofiche nei sistemi real-time.

Questo incidente ha portato a una maggiore attenzione nella validazione e verifica dei software nei sistemi avionici, con l'adozione di standard più rigorosi per lo sviluppo di software safety-critical, come DO-178C. Inoltre, ha rafforzato l'importanza dell'analisi statica per rilevare potenziali vulnerabilità legate alla gestione dei tipi di dati [13].

3.3 Therac-25

Un altro esempio drammatico di fallimento software è il caso del Therac-25, un acceleratore lineare per la radioterapia sviluppato dalla Atomic Energy of Canada Limited (AECL) e utilizzato tra gli anni '80 e '90. Tra il 1985 e il 1987, il dispositivo fu responsabile di sei incidenti gravi, nei quali i pazienti furono esposti a dosi centinaia di volte superiori al valore previsto, portando a ustioni gravi e, in alcuni casi, alla morte.

Il problema principale risiedeva in un bug nel software di controllo, scritto in assembly per il sistema PDP-11.
Questo codice, derivato da versioni precedenti dei Therac-6 e Therac-20, non era stato adeguatamente testato per le modifiche apportate nel Therac-25.

Uno dei bug più critici riguardava la gestione della concorrenza. Il software utilizzava variabili globali per tracciare lo stato del macchinario, ma senza un adeguato meccanismo di sincronizzazione. Questo permetteva a un operatore di inserire rapidamente comandi sulla console, modificando lo stato del sistema prima che i controlli di sicurezza venissero completati.

Ad esempio, un operatore poteva selezionare una modalità di trattamento e correggerla subito dopo. Il software, tuttavia, non gestiva correttamente il cambio di stato e lasciava attiva la modalità precedente, portando a una configurazione errata della macchina. Di conseguenza, il sistema poteva erogare una dose massiccia di radiazioni invece di una quantità controllata.

Inoltre, il manuale fornito insieme alla macchina non conteneva informazioni riguardo i codici degli errori che essa a volte mostrava e questo ha portato gli operatori ad ignorare tali messaggi.

Questo incidente mostrò l'importanza dell'informare l'utente finale sulle procedure da seguire in caso di errori e di rendere la sicurezza parte fondamentale dei processi previsti dall'ingegneria del software, in modo da avere sistemi sicuri "by design" [11].

4. APPROCCI RISOLUTIVI

Gli incidenti storici appena descritto hanno fatto in modo che al giorno d'oggi esistano numerosi approcci diversi al problema della safety, ciascuno con pro e contro. In questa sezione verranno descritte le caratteristiche principali di alcuni di essi.

4.1 Analisi Statica

Una branca dell'informatica che sta acquisendo sempre più rilevanza è quella che riguarda l'analisi dei programmi: conoscere certe proprietà e comportamenti di un programma prima che venga eseguito (**analisi statica**) o durante la sua esecuzione (analisi dinamica) può mettere in condizione di poter fornire garanzie circa il suo consumo di risorse, la sua correttezza, la sua sicurezza o la sua l'efficienza. Queste garanzie, in alcuni ambiti critici, possono risultare imprescindibili ma riuscire a dimostrare queste proprietà non è un'attività banale a causa delle limitazioni teoriche su cui di basa l'informatica stessa.

Uno dei più grandi limiti posti dalle fondamentali intuizioni di Alan Turing riguardo l'analisi dei programmi si evidenzia nell'*halting problem* [16]: il quesito "è sempre possibile, descritto un programma e un determinato input, stabilire se il programma in questione termina o continua la sua esecuzione all'infinito?" è *non decidibile*. Si dice *indecidibile* un problema per il quale non esiste alcun programma che possa determinare, per ogni possibile input, se la risposta sia positiva o negativa in un tempo finito.

Questo risultato porta ad esempio ad intuire come sia inoltre impossibile calcolare l'intera *semantica concreta* di un programma, infatti, essa rappresenta l'insieme (infinito) di tutti i possibili flussi di esecuzione in tutti i possibili ambienti di esecuzione (Figura 1).

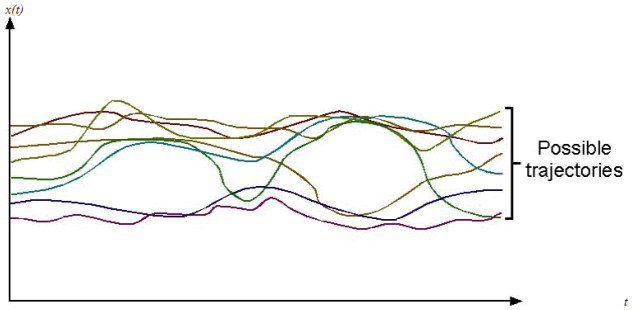

Figure 1: Rappresentazione delle possibili esecuzioni di un programma tramite curve che mostrano l'andamento dei valori in input, dello stato e dell'output in funzione del tempo.

Posti questi limiti, esistono comunque alcuni modi per affrontare il problema della verifica della proprietà di *safety* che risulta essere tra quelle di maggiore interesse. Per cercare di raggiungere tale obiettivo l'analisi dinamica considera direttamente la semantica concreta cercando di stabilire se una qualsiasi delle sue traiettorie intersechi una zona rappresentante uno stato di errore (Figura 2).

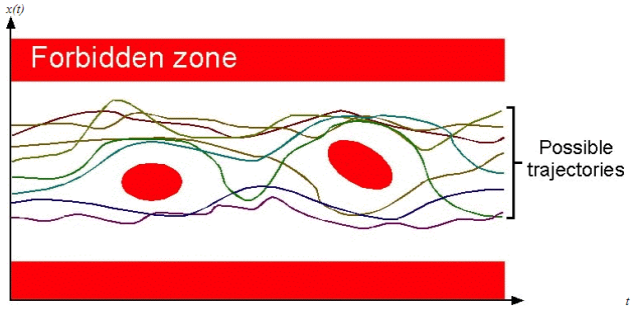

Figure 2: La *forbidden zone* rappresenta gli stati in cui l'esecuzione del programma è in errore.

Tuttavia, la maggior parte delle tecniche che si concentrano sull'analisi a *runtime* soffrono del problema dell'*Absence of Coverage*: a causa dalla natura infinita della semantica concreta, non è possibile assicurare la proprietà di *safety* perché, con questo approccio, verranno sicuramente tralasciati dei possibili flussi di esecuzione che potenzialmente potrebbero portare ad uno stato di errore.

In certi ambiti non critici questo approccio può risultare sufficiente ma, dove tale proprietà è di cruciale importanza, non è possibile lasciare spazio ad incertezze. Perciò, la branca della verifica del software che sembra più adatta agli ambiti critici risulta essere l'analisi statica, approccio speculare rispetto a quello appena visto, che riesce ad abbassare il livello di incertezza sotto le più stringenti specifiche tramite "*approssimazione*".

4.1.1 *Interpretazione Astratta*

L'**Interpretazione Astratta** [8] è un metodo formale dell' analisi statica che consiste nel considerare un sovrainsieme di tutti i possibili flussi di esecuzione di un programma (detto *semantica astratta*) in modo che, se una proprietà risulta verificata per esso, lo sarà sicuramente anche per l'interezza della semantica concreta Figura 3.

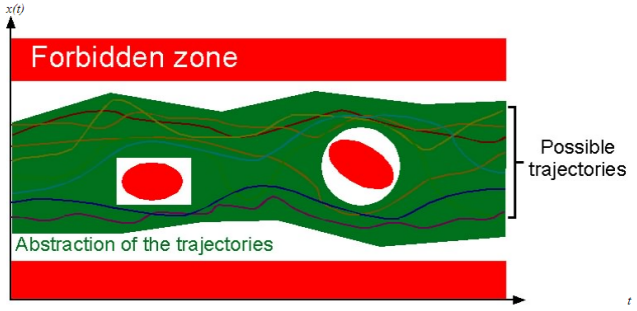

Figure 3: L'area verde rappresenta la *semantica astratta*: un'approssimazione per eccesso della *semantica concreta*.

I due rischi principali della definizione della semantica astratta risiedono nel processo di approssimazione. Infatti, è possibile infrangere la proprietà di *soundness* effettuando un'approssimazione eccessivamente restrittiva o, all'opposto, avere falsi positivi durante l'analisi a causa di un' approssimazione troppo ampia che interseca la zona di errore nonostante il programma non possa in alcun modo raggiungere quello stato (falso positivo).

Nonostante le limitazioni, questo approccio risulta essere effettivamente utilizzato, ad esempio attraverso l'analizzatore statico Astrée [1], da aziende come Airbus, Bosch, ESA ed altri per rispettare gli standard di sicurezza come il sopracitato DO-178C [2].

4.2 Sottoinsiemi e sovrainsiemi di linguaggi

Nonostante il giovane linguaggio di programmazione `Rust` abbia convinto persino gli sviluppatori del kernel Linux [14], ci sono state negli anni diverse proposte riguardo alla creazione di sotto o sovra insiemi safe di linguaggi di programmazione non memory-safe come `C` e `C++`.

4.2.1 *MISRA*

MISRA (Motor Industry Software Reliability Association) è un'organizzazione internazionale nata nel 1994 con l'obiettivo di promuovere lo sviluppo di software sicuro e affidabile, inizialmente nel settore automobilistico. Oggi, le sue linee guida e standard sono ampiamente adottati in vari settori industriali, tra cui aerospaziale, difesa, medicale e telecomunicazioni. L'organizzazione opera attraverso un processo collaborativo, coinvolgendo esperti del settore, aziende e istituzioni per sviluppare e aggiornare i suoi standard.

Questi standard forniscono regole e direttive per la scrittura di codice `C` e `C++` robusto e sicuro. Infatti, essi sono progettati per ridurre il rischio di errori nel software, migliorandone la qualità e la manutenibilità attraverso il divieto di utilizzare funzionalità del linguaggio considerate pericolose nonostante siano permesse. Le linee guida MISRA sono particolarmente utili in contesti in cui il software è soggetto a normative severe, come nei sistemi embedded o in applicazioni safety-critical [12].

Ad esempio, un programma che utilizza una variabile non precedentemente inizializzata infrange il regolamento MISRA nonostante sia contemplato dallo standard `C/C++`.

4.2.2 *C-rusted*

C-rusted propone un approccio alternativo all'abbandono del linguaggio `C` verso linguaggi che offrono garanzie maggiori in termini di safety. L'idea è quella di utilizzare i concetti che si sono rivelati veramente utili in altri linguaggi, come l'**ownership** di `Rust`, e implementarli in una versione safe, secure e energy-efficient di `C`.

Questo avviene tramite annotazioni: il codice di base sarà comunque ISO `C` per mantenere la compatibilità con quanto già esistente (punto di forza caratteristico del `C`) ma il programmatore dovrà aggiungere annotazioni tramite macro (ignorate quindi dal compilatore). Esse verranno sfruttate da un analizzatore statico ad hoc che, grazie al contesto aggiunto, da esse riuscirà a garantire importanti proprietà sul codice.

Per come è stato progettato `C-rusted`, anche l'assenza di annotazioni è informativa: in ISO `C` è possibile che un puntatore qualsiasi sia `null` mentre per ottenere questo comportamento in questa versione più sicura sarà obbligatorio specificarlo tramite annotazione appunto [3].

I maggiori benefici che si hanno nel caso si scelga questa strada per scrivere codice sicuro sono:

- non è necessario modificare codice già esistente se non tramite annotazioni.
- non è necessario addestrare programmatori su un intero linguaggio di programmazione nuovo, ma solamente sul sistema di annotazioni.
- i compilatori `C` esistenti funzionano anche per `C-rusted`.
- i tools già esistenti per `C` funzionano anche per `C-rusted`.

4.2.3 *Safe C++*

`Safe C++`, come `C-rusted`, propone un'alternativa all'abbandono di un altrettanto diffuso linguaggio di programmazione a basso livello: il `C++`. Lo fa attraverso un approccio simile ma non basato su macro, ovvero quello di *estendere un sottoinsieme safe* del linguaggio con meccanismi ad-hoc per implementare idee vincenti di memory safety senza l'overhead della garbage collection. Queste estensioni del linguaggio hanno richiesto l'implementazione di un compilatore `C++` (**Circle** [4]) che supportasse i nuovi elementi della sintassi e che effettuasse i nuovi controlli statici richiesti durante la compilazione. Inoltre, si è resa necessaria una riscrittura completa (realizzata solo in parte) della libreria standard del `C++` perché ovviamente basata su codice unsafe: `Safe C++` considera unsafe tutto ciò che può portare ad *undefined behavior*, come l'uso dei puntatori nel suo complesso o l'utilizzo di variabili prima della loro definizione [5].

4.3 Rust

`Rust` è un linguaggio di programmazione moderno progettato per garantire sicurezza e prestazioni senza l'uso di un garbage collector. La gestione della memoria in Rust si basa su tre concetti fondamentali: **ownership**, **borrowing** e **lifetimes**. Questi meccanismi permettono di prevenire errori comuni come *use-after-free*, *memory leaks* e *data races* in ambienti concorrenti.

4.3.1 Ownership

L'**ownership** è il principio cardine della gestione della memoria in `Rust`. Ogni valore ha un proprietario unico (Algoritmo 1) e quando il proprietario esce dallo scope, la memoria viene automaticamente deallocata.

Algorithm 1 Esempio dimostrativo dell'ownership

```
fn main() {
    let s1 = String::from("Ciao");
    let s2 = s1; // proprieta' trasferita a s2

    // println!("{}", s1);  // Errore!
    println!("{}", s2);  // Stampa "Ciao"
}
```

4.3.2 Borrowing

Poiché l'ownership di un valore non può essere trasferita automaticamente, `Rust` consente di prendere in prestito (**borrowing**) i valori attraverso le *references* (`&`) consentendo di accedere ai dati senza cambiarne il proprietario.

`Rust` distingue tra borrowing **immutabile** (Algoritmo 2) e **mutabile** (Algoritmo 3): non si possono avere più riferimenti mutabili simultanei e non ci possono essere riferimenti mutabili se ci sono già riferimenti immutabili a una stessa variabile (Algoritmo 4).

Algorithm 2 Esempio di borrowing immutabile

```
fn main() {
    let s1 = String::from("Ciao");
    let len = calcola_lunghezza(&s1);
    // Prestito di s1 in modo immutabile

    println!("La␣len␣di␣'{}'␣e'␣{}.", s1, len);
}

fn calcola_lunghezza(s: &String) -> usize {
    s.len()
}
```

In questo esempio, `s1` viene prestata in modo immutabile alla funzione `calcola_lunghezza`. Dopo la chiamata, `s1` rimane valida e può essere utilizzata.

Nel seguente esempio invece, `s1` viene prestata in modo mutabile alla funzione `modifica_stringa`, che modifica la stringa aggiungendo ", Mondo!".

Algorithm 3 Esempio di borrowing mutabile

```
fn main() {
    let mut s1 = String::from("Ciao");
    modifica_stringa(&mut s1);
    // Prestito di s1 in modo mutabile

    println!("{}", s1);  // Stampa "Ciao, Mondo!"
}

fn modifica_stringa(s: &mut String) {
    s.push_str(",␣Mondo!");
}
```

Algorithm 4 Esempio di borrowing mutabile quando già immutabile

```
fn main() {
    let mut s = String::from("Ciao");

    let r1 = &s;  // Prestito immutabile
    let r2 = &s;  // Prestito immutabile
    // let r3 = &mut s;  // Errore!

    println!("{},␣{}", r1, r2);
}
```

4.3.3 Lifetimes

I ***lifetimes*** garantiscono a compile-time che le *references* siano sempre valide e non puntino a memoria deallocata.

Algorithm 5 Esempio gestione lifetime

```
fn main() {
    let str1 = String::from("lunga␣stringa");
    let str2 = "xyz";

    let ris = longest(str1.as_str(), str2);
    println!("La␣stringa␣piu'␣lunga␣e'␣{}", ris);
}

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str{
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

In questo esempio, la funzione `longest` restituisce il riferimento alla stringa più lunga. Il ciclo di vita `'a` garantisce che i riferimenti `x` e `y` siano validi per lo stesso periodo.

Questi concetti rendono `Rust` particolarmente adatto per lo sviluppo di sistemi sicuri e applicazioni ad alte prestazioni secondo le guidelines pubblicate già nel 2023 dalla CISA (Cybersecurity & Infrastructure Security Agency), agenzia di rilevanza internazionale facente parte del dipartimento di Sicurezza Interna degli Stati Uniti [7].

5. CONCLUSIONI

La sicurezza e l'affidabilità del software rappresentano sfide fondamentali nel panorama tecnologico moderno, specialmente per i sistemi safety-critical. L'analisi dei principali incidenti informatici ha dimostrato come errori di programmazione, spesso legati alla gestione della memoria e della concorrenza, possano avere conseguenze devastanti. Per mitigare questi rischi, sono state sviluppate diverse strategie, tra cui linee guida per linguaggi esistenti o modifiche safe degli stessi, analizzatori statici in grado di garantire proprietà di safety, nuovi paradigmi di programmazione come quello introdotto da `Rust`. Quest'ultimo, grazie al sistema di ownership, borrowing e lifetimes, fornisce garanzie di memory safety senza la necessità di un garbage collector, rendendolo una scelta promettente per lo sviluppo efficiente di software critico. Tuttavia, l'adozione di `Rust` e di altre metodologie sicure richiede un cambiamento di mentalità e un adeguato addestramento degli sviluppatori. L'evoluzione di questi strumenti e la loro integrazione nei processi di sviluppo potrebbe ulteriormente migliorare la sicurezza del software, rendendo meno probabili gli errori umani e aumentando la resilienza dei sistemi informatici per garantire un futuro tecnologico più sicuro e affidabile.

6. REFERENCE

[1] AbsInt. Company profile. `https://www.absint.com/profile.htm`.

[2] AbsInt. Success stories. `https://www.absint.com/success.htm`.

[3] R. Bagnara, A. Bagnara, and F. Serafini. C-rusted: The advantages of rust, in c, without the disadvantages, 2023.

[4] S. Baxter. Circle C++ with Memory Safety. `https://www.circle-lang.org/site/index.html`.

[5] S. Baxter and C. Mazakas. Safe C++. `https://safecpp.org/draft.html#the-call-for-memory-safety`, 2024.

[6] C. Calin. Almost 1 billion dollars lost by nintendo in 2007. 008. [Accessed 29-01-2025].

[7] CISA. Secure-by-Design. `http://cisa.gov/resources-tools/resources/secure-by-design`, 2023.

[8] P. Cousot. Abstract Interpretation in a Nutshell. `https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`.

[9] M. Dan. Microsoft excel's bloopers reel: nearly 40 years of spreadsheet errors - whale-sized mistake. 024. [Accessed 29-01-2025].

[10] L. J.L. N° 33-1996: Ariane 501 - presentation of inquiry board report. `https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report#:~:text=A%20detailed%20account%20is%20given,seconds%20after%20lift%2Doff`, 1996. [Accessed 30-01-2025].

[11] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. `https://www.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/therac25.pdf`, 1993.

[12] MISRA. MISRA. `https://misra.org.uk/`.

[13] RTCA. DO-178C: Software considerations in airborne systems and equipment certification. Technical report, Radio Technical Commission for Aeronautics, 2011.

[14] J. Salter. Linus Torvalds weighs in on Rust language in the Linux kernel. 021.

[15] M. Team. A proactive approach to more secure code. `https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/`, 2019. [Accessed 29-01-2025].

[16] A. Turing. On computable numbers, with an application to the entscheidungs problem. *Proceedings of the London Mathematical Society*, 2(42):230–42, 1936.