Security Properties of Different Programming Languages: C, Java, and Python

Teju Kanaparthy (`tk245@duke.edu`), Myles Bell (`mdb125@duke.edu`)

I. Introduction

When programming languages first began to develop, security requirements and considerations of system attacks were not a core design priority. Instead, designers focused primarily on functionality, performance, and usability. However, this approach and attitude within developing programming languages in turn means that, while adversaries continue to develop new techniques to both infiltrate and attack systems, the underlying languages utilized to write such programs remain prone to the same vulnerabilities that they had been designed with at conception. As a result, contemporary systems have been pursuing the mitigation of risks rooted not just in flawed code, but also within the languages themselves.

Specifically, the security properties of a programming language are subject to the synergy of the language's design, runtime architecture, developer practices, and tooling ecosystems. Language-based protections such as memory safety, type safety, thread safety, sandboxing, and access control are among the most critical areas in evaluating whether a language helps enforce or inadvertently weaken security goals of the applications they comprise.

However, a key consideration of the extent to which security is prioritized within the design of a programming language is also heavily dependent on the goals of the language. For instance, low-level languages like C sacrifice memory safety for performance, exposing developers to vulnerabilities like buffer overflows and dangling pointers as these risks represent the trade-offs that accompany a balance between control, performance, and safety [1]. Too often, the integration of a heavy adherence to security is subject to intensive real-time processing that reduces flexibility and developer speed [2]. However, programming languages with such goals do exist today, such as Java, which was explicitly engineered to offer both strong static checks and dynamic runtime enforcement with features like the Java Virtual Machine, byte-code verification, and Java Security Manager [3]. Even with the presence of security-focused implementations of programming languages, it is essential to recognize that developer practices, third-party packages, and other elements of the system all play an equally relevant role in how vulnerable a system will be to breaches and cyberattacks. Therefore, though our discussion of the security of the inherent design of programming languages is paramount, vigilant developer practices and usage must complement software languages in order to rigorously establish the system's overall resilience against exploitation.

This report will trace the evolution of programming language security by exploring three major languages—C, Java, and Python—through the lens of their design philosophies, runtime guarantees, and inherent vulnerabilities. With this understanding, we then address how security has transitioned from a remedial afterthought, to now a central concern in language development. We conclude by examining future trends, including the rise of memory safe alternatives like Rust, the push for more secure development workflows, and the emerging ecosystem of tools, packages, and language extensions aimed at enforcing safety and privacy policies by design. By learning the inner-workings of the security properties of these prominent languages as well as their relation to the implementation of other tools and aspects of ensuring program safety, we gain a more holistic understanding of how language design and developer measures intersect to shape the overall security posture of different modern software systems.

II. C

In the early 1970s, Dennis Ritchie developed the C programming language at Bell Labs with the intention of implementing operating systems, such as UNIX, to be used by trusted developers in controlled environments. Its goals as a language involved prioritizing efficiency, low-level memory access, and portability across hardware architectures. However, as attackers and the internet continued to evolve over time, networks of hostile and sophisticated threats have emerged. As a result, many of C's powerful and efficient features have since also served as its greatest liabilities, and security has now become a core concern within its design.

Notably, C allows programmers direct access to memory using pointers and permits pointer arithmetic, which is the ability to perform operations on memory addresses. Such privileges allow developers to obtain high performance and control, but it also simultaneously removes safety checks that other languages include, such as automatic bounds checking, no built-in garbage collection, and no protection against common mistakes such as accessing freed memory or dereferencing null or uninitialized pointers. Additionally, C isn't a type-safe or memory-safe language, which makes it directly vulnerable to a wide range of memory-related attacks. According to an analysis of security issues reported through Stack Overflow, the most predominant exploits are buffer overflows, use-after-free errors, integer overflows, and format string vulnerabilities [4].

A buffer overflow is a vulnerability that occurs in C when a program writes more data into a fixed-size buffer than it can hold, which can then overwrite adjacent memory. Attackers often exploit this condition to inject and execute malicious code. Without C having built-in protections, developers are the ones that must implement their own input validation and bounds checks to avoid such errors. Similarly, use-after-free bugs occur when a program continues to use memory after it has been deallocated, which can then result in undefined behavior or unwanted remote code execution. To address these long-standing challenges, researchers at the Software Engineering Institute at Carnegie Mellon University developed the SEI CERT C Coding Standard. This standard outlines a set of secure coding rules and best practices to help developers avoid unsafe constructs. It offers alternatives to common insecure patterns by replacing vulnerable library functions with safer variants and enforcing stronger input validation and error handling procedures. However, though this standard outlines how to write C code to avoid vulnerabilities, this does not alter the language's core susceptibility to memory-related flaws, and is heavily dependent on developer responsibility and diligence, making it neither trustworthy nor enforceable as a guaranteed defense.

As a result, to supplement these coding guidelines, C employs runtime defenses at the compiler and operating system levels. For instance, stack canaries are values placed between buffers and control data in memory. If a buffer overflow occurs, the canary is overwritten, which triggers detection before the function returns and prevents code execution [5]. Moreover, C programs can employ the feature of address space layout randomization, which essentially moves memory regions like the stack and heap to unpredictable locations so that attackers are unable to easily target a specific address. C also allows data execution prevention, which marks certain regions of memory as non-executable and, in turn, blocks code injection attacks from executing. These mitigation strategies, along with static analysis, fuzz testing, and runtime instrumentation, form the modern security toolkit for building and maintaining secure C applications, but they are often add-ons rather than guarantees baked into the language itself [6].

Despite these risks, C remains fundamental in systems programming as it provides the low-level access needed for writing high-performing operating systems, device drivers, and real-time software. Security trade-offs should be consciously balanced against performance requirements, legacy compatibility, and stakeholder priorities [7]. As a result, C's irreplaceable and powerful performance and control can outweigh the corresponding security trade-offs as long as developers adhere to best security practices within C's integration into modern software ecosystems.

III. Java

Introduced in 1995 by Sun Microsystems, Java revolutionized programming language design by emphasizing security, portability, and reliability over direct hardware control. In comparison to lower-level languages like C, Java consists of a comprehensive security architecture built directly into the language as well as its runtime environment. Java also utilizes a layered and systematic approach to achieving language-based security, consisting of four major components: language-level safety features, bytecode verification, a controlled class loader architecture, and a configurable runtime security manager. Together, these layers work to prevent unauthorized behavior and enforce strict separation between trusted and untrusted code during execution.

To begin, Java is a memory-safe language, as it does not allow pointer arithmetic and direct memory access. Instead of relying on the programmer to manage memory manually, Java equips an automatic garbage collector, which safely frees memory as soon as it's no longer in use. Therefore, due to these properties and this structure of Java, common errors such as buffer overflows, use-after-free, or double-free bugs are not a potential concern. Additionally, Java's typing system enforces type compatibility both at compile time and runtime by default, which can be observed through the exceptions raised when violations occur (e.g., `ClassCastException`). These characteristics exemplify that limiting the bounds of what is syntactically and semantically possible is directly responsible for reducing the risk of unsafe behavior and making Java inherently more secure.

Java programs are also designed to compile into an intermediate representation called bytecode, which is executed by the Java Virtual Machine (JVM). The JVM serves as an abstract computing environment that emulates a processor, as it sandboxes the execution of Java programs by equipping a bytecode verifier that inspects each class for violations of type safety, stack discipline, and illegal operations. As an additional standard of protection, access to the computer system's files, networks, and more is determined through runtime libraries and APIs that act in compliance with Java's internal rules on resource usage. This layered control is further reinforced by the SecurityManager, which managed runtime permissions for elements such as Java applets, where untrusted code embedded in web pages was executed within this constrained sandbox under various security and access-control policies [3].

Yet, Java is not entirely resistant to security vulnerabilities. Java applications remain susceptible to logic-level vulnerabilities like SQL injection in the event that user input isn't properly sanitized. Additionally, many of the threats it does face arise from how it's used and extended, rather than from flaws in the language itself. For example, Java allows the use of the Java Native Interface (JNI) to link with lower-level native code, such as C libraries. This can reintroduce the very vulnerabilities of bypassing type and memory safety guarantees that Java was designed to avoid, as security is only as strong as the weakest link in the software stack [6]. Another prominent actor in the introduction of vulnerabilities is often the developer, as many compromise safety for convenience by disabling hostname verification, TLS verification, and SSL certificate checks, which essentially trusts all certificates and exposes their applications to man-in-the-middle attacks [8]. These patterns reinforce the true importance of secure coding practices, even while using a safe language.

IV. Python

Introduced in 1991 by Guido van Rossum, Python was a minimalist programming language designed to prioritize rapid development and accessible syntax that's readable by humans. Similar to Java, Python is a high-level, dynamically-typed language that abstracts away low-level concerns like memory management which allows it to have multi-purpose usage, such as within web development, scientific computing, and machine learning. However, this same abstraction means that Python was not originally engineered with rigorous security enforcement at its core, but rather with usability and extensibility.

Python avoids many of the classic memory issues seen in lower level languages like C. It does not allow pointer arithmetic, and it uses automatic garbage collection, thereby eliminating the need for manual memory deallocation and reducing risks like buffer overflows or use after free vulnerabilities. At runtime, Python enforces type safety, which is code that attempts to operate on incompatible types results in a `TypeError` or similar exception in order to mitigate memory corruption and type confusion attacks.

However, since Python's dynamic nature delays type checking and binding until runtime, certain bugs and vulnerabilities can be undetected during development, and instead surface during execution. As a result, Python becomes vulnerable to logic-based exploits, like command injection or inadequate input validation within its applications [9].

Additionally, it's important to address that, unlike Java's JVM sandbox or C's reliance on OS level containment, Python lacks an official, built-in sandbox model. Early attempts to introduce restricted execution modes include rexec and Bastion, but these efforts were eventually deprecated due to inherent design flaws that made them unreliable. As a result, the safe execution of untrusted code with Python must be executed in external environments, such as virtual machines or third-party tools for sandboxing.

In terms of security, Python has an increased potential for supply-chain vulnerabilities due to its high extensibility via third party packages in order to support its applications that often depend on many external modules. Outdated libraries consequently can introduce security flaws into otherwise safe codebases, which must be mitigated through developers following best security practices for dependency management, such as by utilizing tools like pip audit to identify known common vulnerabilities and exposures in package dependencies.

Ultimately, Python's security properties reflect the trade-offs of its original design. Its flexibility and simplicity enable its rapid development, but these same traits leave it reliant on developer responsibility and system tools in order to truly enforcing robust security. Ultimately, Python exhibits how language design choices that prioritize usability and developer convenience over structural enforcement can undermine native security, which then highlights the risks of leaving out safeguards against untrusted code and vulnerable dependencies. Without robust, language-enforced guarantees, true protection must instead be architected externally through containment, secure tooling, and proper development practices.

V. Comparative Security Implications

It's important to acknowledge that security is one among many competing priorities, such as usability, performance, portability, and compatibility with legacy systems, that programming languages have as their design objective. The security of a programming language is shaped by the language's purpose, its runtime architecture, the historical context of its design, and the priorities of its stakeholders, which includes developers, users, and system designers [7]. While languages like C, Python, and Java are frequently compared in terms of their performance or syntax, a meaningful comparison for each of these languages' security is established only when we consider the trade-offs that determine their behavior in real-world systems.

A language like C was built for speed and direct hardware control, sacrificing memory safety in favor of performance and system-level access. In contrast, Java was explicitly designed to prevent memory corruption by removing pointer arithmetic and introducing managed memory via garbage collection [3]. Python, designed for rapid application development and dynamic scripting, prioritizes flexibility over static guarantees, so it enforces type safety and memory safety through runtime checks [2].

These design choices result in languages exhibiting distinct vulnerability profiles. C, being an unsafe language, grants programmers unrestricted access to memory via pointers and lacks both array bounds checking and automatic memory management, resulting in well-documented vulnerabilities such as buffer overflows, use-after-free, and integer overflows [1]. Java and Python, on the other hand, prevent direct memory manipulation and restrict undefined behaviors through runtime checks. Java raises exceptions like `ArrayIndexOutOfBoundsException` and `ClassCastException` when safety is violated, while Python raises errors like `TypeError` and `IndexError`. However, these protections come with performance overhead. Python's Global Interpreter Lock (GIL) limits parallelism, as only one thread can execute Python bytecode at a time [2], while Java's garbage collector and runtime bytecode verification introduce delays compared to compiled native code [4].

These languages also diverge in the scope and enforcement of access control, as Java features a multi-layered security model, with Java's Security Manager enforcing runtime access control policies, denying untrusted code permissions such as file access or network I/O, whereas a C program has full access to the process's privileges and must be sandboxed externally using OS-level tools like Seccomp or `chroot`. Finally, Python offers some limited controls, but past efforts to implement a secure sandboxed Python runtime have failed due to its inherently introspective nature and mutable state [3].

The issue of responsibility also varies by language. C places nearly all responsibility on the developer, including the manual implementation of memory management, thread safety, access control, and error handling. Java and Python offer more protections; however, they still require the developer to understand and use APIs securely [8]. This distinction between language-level security and developer-level responsibility underscores that secure software is not just a product of a secure language, but also of secure practices.

Further complicating language security are the varying tool ecosystems available for vulnerability mitigation. In Java, static analyzers like SpotBugs and security-typed extensions like Jif help detect logic flaws or enforce information flow policies [7]. C relies heavily on external static analysis tools such as Coverity and Clang's analyzer, fuzzing, and runtime mitigation techniques like stack protection, DEP, and code signing [6]. Python developers often depend on dynamic checks and package-level security updates, but are more susceptible to supply chain attacks due to Python's reliance on third-party packages via `pip`.

Ultimately, no language is inherently secure as these implications of security stem from the design philosophy and use-case priorities embedded into each language. The main priority is achieving the desired balance between a language's features, its security guarantees, and the developer's intended use from a full-stack perspective, which includes considerations of bytecode versus native compilation, access control models, thread safety and type systems, runtime patching, and mitigation tooling.

According to studies by Shahriar and Zulkernine, the most effective strategy involves combining static analysis (which inspects code without executing it) and dynamic analysis (which observes behavior at runtime), since static tools can identify issues early in development but may produce false positives, while dynamic tools, such as fuzzers, detect vulnerabilities only during execution [10]. Such hybrid analysis is able to maximize and enhance coverage by evaluating security across dimensions like completeness, soundness, and inference types.

VI. Further Advancements

While the dominant programming languages today remain entrenched due to ecosystem maturity and widespread familiarity, recent advancements are reshaping how security can in fact be embedded more directly into language design. Languages like Jif and Laminar were developed specifically to enforce information flow policies. Jif, which is an extension of Java, enables compile-time enforcement of confidentiality and integrity constraints through security-typed annotations. Similarly, Laminar integrates secrecy and integrity labels into its type system, thereby offering refined data flow control. However, these languages face barriers to widespread adoption due to steep learning curves, limited compatibility with existing software, and difficulty integrating with prominent legacy codebases [2]. Another effort includes Cyclone, which was supposed to be a type-safe derivative of C striving to preserve low-level control while avoiding buffer overflows and other common pitfalls in C code. Despite its attempts to retrofit safety into C's capabilities, it was ultimately discontinued due to practical limitations, such as poor developer uptake and performance overhead compared to C [6].

More recently, research has turned toward leveraging machine learning to detect vulnerabilities in high-level languages. Zhao et al. (2024) introduced a deep learning approach using CodeBERT, a transformer model pre-trained on both natural and programming languages. Their system significantly outperformed previous LSTM-based models by better understanding semantic structures in Python source code. The method achieved up to 98% classification accuracy across a range of vulnerability types including SQL injection, remote code execution, and insecure encryption, suggesting a path forward for smarter, context-aware vulnerability scanning [11].

Ultimately, the security properties of a programming language are shaped by an intersection of design philosophy, runtime architecture, tooling, and user responsibility. Java and Python offer more built-in safety features, but still rely heavily on correct developer behavior and safe integration practices. C's undeniable power comes with high risk and requires compensatory measures at the compiler, runtime, and OS levels. As language security evolves, the goal is not to abandon older languages and systems, but to better equip developers to leverage them safely through education, tool support, and refined practices.

References

[1] S. Bocetta, "How secure is java compared to other languages?" 2020, javaWorld. [Online]. Available: https://www.proquest.com/docview/2393239562

[2] C. Hamilton, "Security in programming languages," https://www.cs.tufts.edu/comp/116/archive/fall2015/chamilton.pdf, 2015, tufts University, COMP 116 Course Paper.

[3] A. Sterbenz, "An evaluation of the java security model," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2000, pp. 66–79. [Online]. Available: https://doi.org/10.1109/SECPRI.2000.848446

[4] R. Croft, Y. Xie, M. Zahedi, M. A. Babar, and C. Treude, "An empirical study of developers' discussions about security challenges of different programming languages," *Empirical Software Engineering*, 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-10054-w

[5] A. Ballman and D. Svoboda, "Avoiding insecure c++: How to avoid common c++ security vulnerabilities," in *2016 IEEE Security Development Conference (SecDev)*. IEEE, 2016, pp. 65–72.

[6] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," *ACM Computing Surveys*, vol. 44, no. 3, pp. 11:1–11:46, 2012. [Online]. Available: https://doi.org/10.1145/2187671.2187673

[7] G. Elahi and E. Yu, "A goal-oriented approach for modeling and analyzing security trade-offs," in *Conceptual Modeling – ER 2007*, ser. Lecture Notes in Computer Science, C. Parent, K.-D. Schewe, V. C. Storey, and B. Thalheim, Eds. Springer, 2007, vol. 4801, pp. 375–390. [Online]. Available: https://doi.org/10.1007/978-3-540-75563-0_26

[8] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 372–382. [Online]. Available: https://doi.org/10.1145/3180155.3180201

[9] F. C. G. Bogaerts, N. Ivaki, and J. Fonseca, "A taxonomy for python vulnerabilities," *IEEE Open Journal of the Computer Society*, vol. 5, pp. 1–15, 2024.

[10] C. Salka, "Programming languages and systems security," *IEEE Security & Privacy*, vol. 3, no. 3, pp. 80–83, 2005. [Online]. Available: https://doi.org/10.1109/MSP.2005.77

[11] K. Zhao, S. Duan, G. Qiu, J. Zhai, M. Li, and L. Liu, "Python source code vulnerability detection based on codebert language model," in *2024 7th International Conference on Algorithms, Computing and Artificial Intelligence (ACAI)*. IEEE, 2024, pp. 1–8, ©2024 IEEE.