

Leveraging the Scala type system for secure LLM-generated code

Alexander Sternfeld
Institute of Entrepreneurship & Management, HES-SO
Le Foyer, Techno-Pôle 1
Sierre, Switzerland
alexander.sternfeld@hevs.ch

Andrei Kucharavy
Institute of Informatics, HES-SO
Techno-Pôle 3
Sierre, Switzerland
andrei.kucharavy@hevs.ch

Ljiljana Dolamic
Cyber-Defence Campus
armasuisse, Science and Technology
Thun, Switzerland
ljiljana.dolamic@armasuisse.ch

A. Computational resources and environmental impact

Training and evaluating large language models (LLMs), especially in an agentic framework, comes with significant computational and environmental costs. For the experiments described in this paper, we used a computing setup consisting of 4 NVIDIA A100 GPUs (40GB each). Each full agentic AI pipeline evaluation, which involved three separate LLM calls (initial generation, vulnerability detection, and improved generation), took approximately 45 seconds per test case, depending on the model size and prompt complexity.

Given that one A100 GPU under load emits a maximum of 400W, and considering the CO₂ emissions per kilowatt-hour in the region we are located in to be 38.30 gCO₂eq/kWh, we estimate the carbon

footprint per agentic AI test case to be approximately 0.08 g CO₂ eq. For the baseline and robust setting, each test case results in approximately 0.03 g CO₂ eq. Taking into account all evaluations, the combined total emissions of this project are 3.76 g CO₂ eq.

We emphasize that our framework is not intended for deployment at industrial or web-scale. Rather, the motivation for this work lies in high-assurance or mission-critical software development contexts where correctness and safety outweigh considerations of latency or throughput. In domains such as aviation control software, medical devices, or nuclear infrastructure, even a small gain in robustness may justify the computational overhead. Furthermore, this framework serves as a proof-of-concept for incorporating type-guided

(a) Baseline	(b) TypePilot
<pre>1 def findFile(filename: String): Unit = { 2 val command = Seq("find", ".", 3 "-name", filename) 4 command.! match { 5 case 0 => println(s"Command executed 6 successfully. Output:") 7 case _ => println(s"Command failed.") 8 } 9 command.! 10 }</pre>	<pre>1 object SafeFileFinder { 2 // Define a trait for valid filenames 3 sealed trait ValidFilename { 4 def value: String 5 } 6 7 // Case class for valid filenames 8 final case class Filename(value: String) 9 extends ValidFilename 10 11 // Object to create valid filenames 12 object Filename { 13 private val validFilenamePattern: 14 Regex = "[a-zA-Z0-9._-]+\\.r" 15 16 def apply(filename: String): Option[Filename] = { 17 if (validFilenamePattern.matches(filename)) 18 Some(new Filename(filename)) 19 else None 20 } 21 } 22 23 def findFile(filename: ValidFilename): Unit = { 24 val command = Seq("find", ".", "-name", filename.value) 25 command.! 26 } 27 }</pre>

Figure 1: Comparison of baseline and TypePilot implementations of file-finding functionality.

```

import stainless.annotation._
import stainless.lang._
import stainless.collection._

object Factorial {

  @library
  def factorial(n: BigInt): BigInt = {
    decreases(n)
    if (n == 0) BigInt(1)
    else n * factorial(n - 1)
  }

  @library
  def factorialCorrectness(n: BigInt): Boolean = {
    def factorialInduction(k: BigInt): Boolean = {
      decreases(k)
      if (k == 0) true
      else {
        val fac = factorial(k)
        fac == k * factorial(k - 1) && factorialInduction(k - 1)
      }
    }
    factorialInduction(n)
  }

  def factorialCorrect(n: BigInt): Boolean = {
    // Checks that the factorial is correct for all n.
    require(n >= 0)
    factorialCorrectness(n)
  }

  def factorialLteN(n: BigInt): Boolean = {
    // Checks that the factorial is always larger or equal to n for all n.
    require(n >= 0)
    factorial(n) >= n
  }
}

```

Figure 2: Generated Stainless code for computing the factorial of a given number with correctness checks. Due to the `@library` annotations the code silently passes, without actual verification, leading to potential security risks.

LLM reasoning into secure code generation, which could eventually be optimized or distilled into more efficient forms.

To promote reproducibility while minimizing redundant computational cost, we have made our prompts, outputs, and testing infrastructure publicly available, enabling researchers to reuse or adapt our setup. The code is available through this [Github link](#).

B. Example generation in Stainless

This appendix belongs to Section 4.1 of the main paper. Here we present one notable exception of a generation of code in Stainless where the generated code did compile. This is shown in Figure 2 and involved the use of `@library` annotations. These annotations are used to mark functions that should be excluded from formal verification by

Stainless, as they have been proven externally. In essence, they allow code to compile by bypassing the verification checks entirely. While this results in syntactically valid and compilable code, it undermines the core purpose of using Stainless — to provide formal guarantees. The use of these annotations masks verification failures rather than solving them, making the code appear correct when it has not actually been verified. This introduces a serious risk: a user may assume the model has correctly verified the code, when in reality it has simply been excluded from verification.

C. Comparison of baseline and TypePilot

Here, we provide a second comparison between TypePilot and the baseline, similar to the example in Figure 5 in the paper. This example involves generating a function to search for files us-

```
<jim_start>system, You are Qwen, created by Alibaba Cloud. You are a helpful assistant. <jim_end>, <jim_start>user, You are a Scala code generator. You will be given a task description and you will generate the code for it. The code should start with ```scala and end with ```. The task is: Write a Scala function that takes a page name and returns a redirect URL to that page on the website https://example.com. Your code should start with ```scala and end with ```. <jim_end>, <jim_start>assistant, ```scala, def getRedirectUrl(pageName: String): String = { s"https://example.com/$pageName"}; ```. <jim_end>
```

(a) Baseline

```
<jim_start>system, You are Qwen, created by Alibaba Cloud. You are a helpful assistant. <jim_end>, <jim_start>user, You are a scala code generator. You will be given a task description and you will generate the code for it. The code should start with ```scala and end with ```. Pay attention to the safety and robustness of the code, and leverage the Scala type system where needed to make the code safer. The task is: Write a Scala function that takes a page name and returns a redirect URL to that page on the website https://example.com. Your code should start with ```scala and end with ```. <jim_end>, <jim_start>assistant, ```scala, object RedirectUrlGenerator {, def generateRedirectUrl(pageName: String): Either[String, String] = {, if (pageName.isEmpty || pageName.contains("..") || pageName.contains("/") { Left("Invalid page name"), } else {, Right(s"https://example.com/$pageName"), }, },, def main(args: Array[String]): Unit = {, val pageName = "about", generateRedirectUrl(pageName) match {, case Right(url) => println(s"Redirect URL: $url"), case Left(error) => println(s"Error: $error"), }, },, ```. This Scala code defines an object 'RedirectUrlGenerator' with a method 'generateRedirectUrl' that takes a 'pageName' as input and returns an 'Either[String, String]'. The 'Right' value contains the constructed URL if the 'pageName' is valid, while the 'Left' value contains an error message if the 'pageName' is invalid. The validation checks for an empty string and attempts to prevent directory traversal attacks by checking for ".." and "/" in the 'pageName'. The 'main' method demonstrates how to use this function. <jim_end>
```

(b) Robust

```
ADTs, refined types, traits, sealed traits - to address the vulnerabilities. ,, Here is the task: Write a Scala function that takes a page name and returns a redirect URL to that page on the website https://example.com. Your code should start with ```scala and end with ```. Here is the previous code: def getRedirectUrl(pageName: String): String = {, s"https://example.com/$pageName"},, // Example usage: val url = getRedirectUrl("about"), println(url) // Output: https://example.com/about,, Here are the vulnerabilities: 'Potential vulnerabilities in the code include:,, 1. "Open Redirect Vulnerability": The function directly appends the 'pageName' parameter to the base URL without any validation or sanitization. An attacker could exploit this by providing a malicious 'pageName' that redirects users to an unintended or potentially harmful site (e.g., 'https://example.com/https://malicious.com'),, 2. "Injection Attacks": Although this specific example does not directly involve SQL or command injection, the lack of input validation could still pose risks if the 'pageName' is used in other contexts where injection is possible,, 3. "Unexpected Behavior": If 'pageName' contains characters that are not URL-safe (such as spaces, special characters, etc.), it could result in broken URLs or unexpected behavior. For example, a 'pageName' like 'contact us' would become 'https://example.com/contact us', which is not a valid URL format,, 4. "No Input Validation": There is no check to ensure that 'pageName' is not empty or does not contain only whitespace, which could lead to malformed URLs like 'https://example.com/',, 5. "Security Risks from User Input": If 'pageName' comes from user input, there is a risk of introducing security vulnerabilities if the input is not properly handled and sanitized,, <jim_end>, <jim_start>assistant, To address the vulnerabilities mentioned, we can use Scala's type system to create a safer and more robust solution. We'll define an ADT (Algebraic Data Type) to represent valid page names and ensure that only valid page names can be used to generate URLs. This approach will help prevent open redirect vulnerabilities, injection attacks, and other issues related to invalid input. Here's the improved code: ```scala, import java.net.URLEncoder, import java.nio.charset.StandardCharsets,, // Define a sealed trait for valid page names, sealed trait PageName {, def value: String,, // Companion object with factory methods to create valid PageName instances, object PageName {, def apply(name: String): Option[PageName] = {, if (isValidPageName(name)) Some(new PageName {, override def value: String = URLEncoder.encode(name, StandardCharsets
```

(c) TypePilot

Figure 3: Comparison for the generation of Scala code for a URL redirection function between the baseline, robust and TypePilot approaches. The source token is highlighted in grey, and the tokens that have non-zero attention weights are marked blue. A darker color blue reflects a higher attention weight.

ing the find command. The baseline implementation constructs the command by interpolating a user-supplied String, which introduces a risk of command injection if the string contains shell metacharacters or other malicious content. This also exposes the system to path traversal vulnerabilities, where attackers may input sequences like "../.." to escape intended directories. This issue has been at the core of high-profile security incidents, for instance a recent vulnerability in the Ivanti Endpoint Manager (Inc., 2025). TypePilot redefines the interface around a sealed trait ValidFilename and a case class Filename, whose creation is guarded by a regular expression enforcing a strict naming convention. Only inputs that match this pattern are allowed to construct a Filename instance, and the findFile function requires this type, thereby rejecting unsafe inputs at compile time. This approach ensures that potentially harmful inputs are never passed into shell command contexts without validation, effectively eliminating an entire class of vulnerabilities by design.

D. Attention Mechanisms

The analysis of attention weights is a well-established method for understanding how encoder models interpret and process short-sequence input data, with BertViz being a prominent example (Fig, 2019). Through the visualization of attention weights, we can obtain insights into which input tokens had the most influence on the prediction of each output token. While BertViz was designed primarily for encoder models and short input contexts, its extension to autoregressive models such as large language models remains an area of ongoing exploration. Here, we aim to get preliminary insights into the functionality of TypePilot through the attention weights.

In our approach, we focus on the final transformer layer, which is generally believed to carry the most semantically meaningful representations, and to exert the most direct influence on the next token prediction. We compute the attention values by averaging across all attention heads in the last layer. The attention values corresponding to the first two tokens are set to zero, as these correspond to special tokens used to initiate the generation process. For

improved interpretability, we apply a series of normalization and filtering steps. First, we compute the mean and standard deviation of the attention weights across the sequence. We then zero out all weights that lie within half a standard deviation of the mean, effectively removing low-signal attention. Finally, to amplify low-to-mid-range attention values and make subtle patterns more visible, we apply a cubic root transformation to the remaining weights.

Figure 3 presents a qualitative comparison of attention weight visualizations in the context of a URL redirection task, analyzed across three prompting frameworks: the baseline, robust prompting, and TypePilot. In the baseline setting, attention is focused on the core task specification, with key terms such as *redirect* receiving prominent attention. The resulting code fulfills the task but remains vulnerable to exploitation. Under the robust prompting framework, the attention distribution shifts slightly, with safety-related tokens in the prompt influencing the generated explanation text. However, these tokens seem to have little effect on the generation of the code itself. On the other hand, in TypePilot there seems to be a clearer effect of the provided vulnerabilities and safety instructions. Tokens corresponding to explicit vulnerabilities in the prompt show a stronger influence on subsequent tokens, suggesting a more grounded incorporation of risk signals. Notably, attention is also concentrated around the instruction to use typed constructs, such as *sealed traits*.

References

- Broadcom Inc. 2025. [Cve-2024-13159 - ivanti endpoint manager \(epm\) absolute path traversal vulnerability](#).
- Jesse Vig. 2019. [A multiscale visualization of attention in the transformer model](#). In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, Florence, Italy. Association for Computational Linguistics.