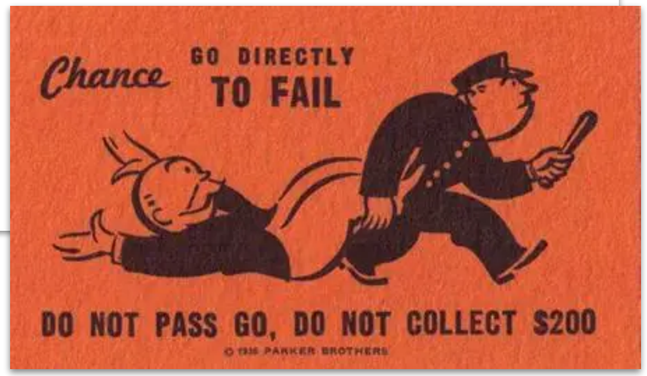
Security Quality: Findings from our Empirical Studies

Coen De Roover
coen.de.roover@vub.be

Gilberto Recupito
gilberto.recupito@vub.be

More famous defects .. in browsers

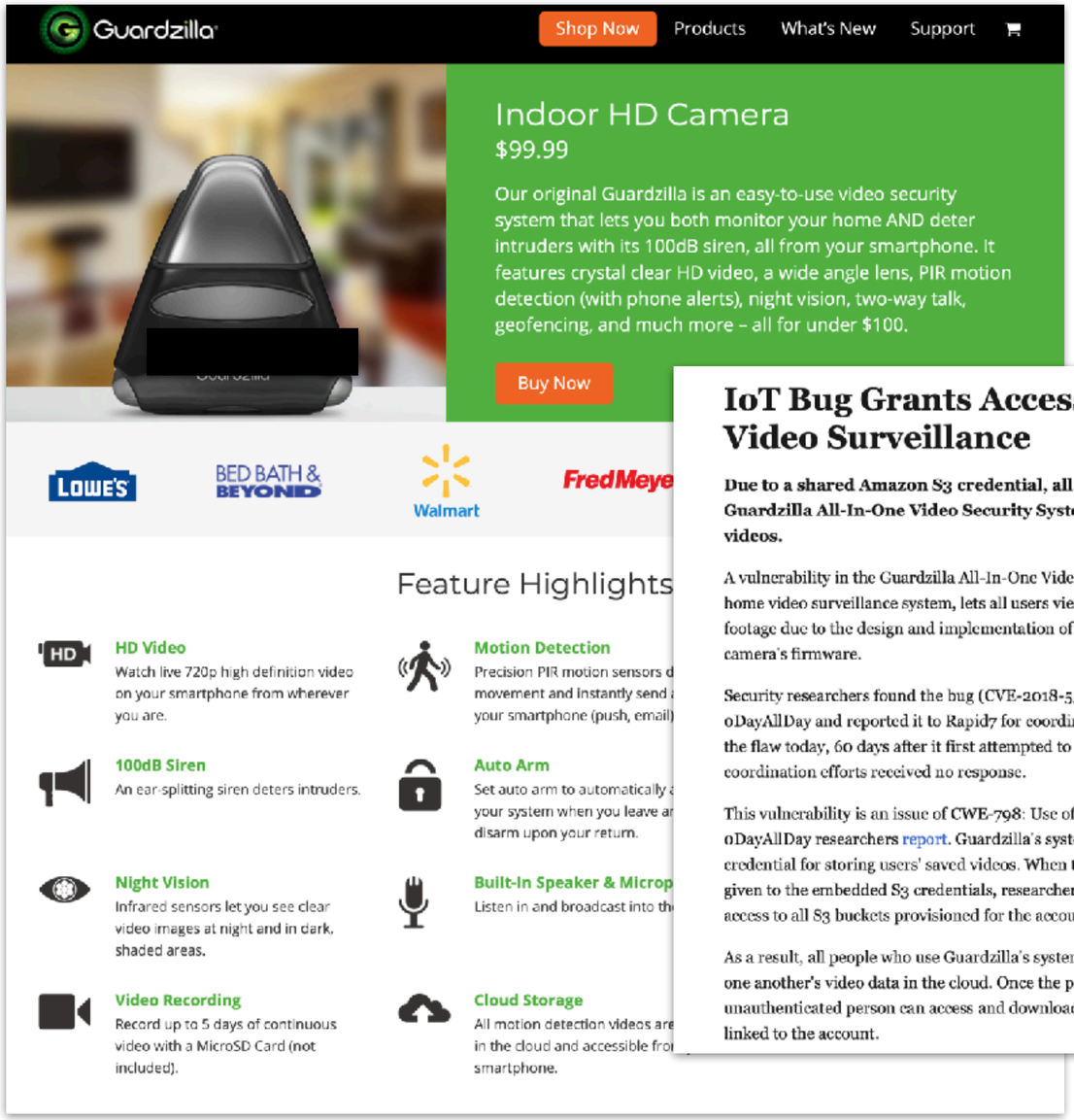
```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
   OSStatus        err;
   ...

   if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
      goto fail;
   if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      goto fail;
      goto fail;
   if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;
   ...

fail:
   SSLFreeBuffer(&signedHashes);
   SSLFreeBuffer(&hashCtx);
   return err;
}
```



On 22nd of February 2014, Apple had to release a **critical security update** for all of its operating systems.

It turns out that signature verification would never fail, not even for arbitrary keys provided by a man in the middle.

More famous defects .. in cameras



IoT Bug Grants Access to Home Video Surveillance

Due to a shared Amazon S3 credential, all users of a certain model of the Guardzilla All-In-One Video Security System can view each other's videos.

A vulnerability in the Guardzilla All-In-One Video Security System, an IoT-enabled home video surveillance system, lets all users view one another's saved surveillance footage due to the design and implementation of Amazon S3 credentials inside the camera's firmware.

Security researchers found the bug (CVE-2018-5560) during an event held by 0DayAllDay and reported it to Rapid7 for coordinated disclosure. Rapid7 published the flaw today, 60 days after it first attempted to contact the vendor. Multiple coordination efforts received no response.

This vulnerability is an issue of CWE-798: Use of Hard-coded Credentials, 0DayAllDay researchers report. Guardzilla's system uses a shared Amazon S3 credential for storing users' saved videos. When they investigated the access rights given to the embedded S3 credentials, researchers found they provide unlimited access to all S3 buckets provisioned for the account.

As a result, all people who use Guardzilla's system for home surveillance can view one another's video data in the cloud. Once the password is known, any unauthenticated person can access and download stored files and videos in buckets linked to the account.

Embedded S3 Credentials Unlimited Access Policy

Once the binaries were extracted from the firmware they were analyzed in IDA Pro to determine if any vulnerabilities could be identified. Once the main.exe had been disassembled and analyzed it was noted that a set of strings resembled AWS credentials:

```
.rodata:002FF5... 00000015  C  AKIAJQDP34RKL7GGV7OQ
.rodata:002FF6... 00000028  C  IgH8yFmmpMbnkcUaCqXJIRIozKVaXaRhE7PWHAYa
.rodata:002FF6... 00000011  C  s3.amazonaws.com
.rodata:002FF6... 00000011  C  motion-detection
```

Following the references, we can see that they are exports from the binary that are labeled: accessKey, secretAccessKey, hostname, and bucket. This format lines up with how AWS bucket keys are designed:

```
.data:00090230                  EXPORT accessKeyIdG
.data:00090230 accessKeyIdG     DCD aAkiajqdp34rkl7     ; DATA XREF: .got:accessKeyIdG_ptr↑o
.data:00090230                                          ; "AKIAJQDP34RKL7GGV7OQ"
.data:00090234                  EXPORT secretAccessKeyG
.data:00090234 secretAccessKeyG DCD aIgh8yfmmpmbnkc     ; DATA XREF: .got:secretAccessKeyG_ptr↑o
.data:00090234                                          ; "IgH8yFmmpMbnkcUaCqXJIRIozKVaXaRhE7PWHAY"...
.data:00090238                  EXPORT hostName
.data:00090238 hostName         DCD a3_amazonaws_c      ; DATA XREF: .got:hostName_ptr↑o
.data:00090238                                          ; "s3.amazonaws.com"
.data:0009023C                  EXPORT bucket
.data:0009023C bucket           DCD aMotionDetectio     ; DATA XREF: aws_video_upload_thread+188↑o
.data:0009023C                                          ; aws_video_upload_thread+194↑r ...
```

AccessKeyIdG	AKIAJQDP34RKL7GGV7OQ
secretAccessKeyG	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
hostName	s3.amazonaws.com
bucket	motion-detection

The following script was developed to test the S3 credentials to determine if they were valid as well as determine what access writes the credentials had:

```
import boto3
# Create an S3 client
s3 = boto3.client('s3',aws_access_key_id='AKIAJQDP34RKL7GGV7OQ',aws_secret_access_key
='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',region_name='us-west-1')

try:
    result = s3.get_bucket_policy(Bucket='motion-detection')
    print(result)
except Exception as e:
    print(e)
```

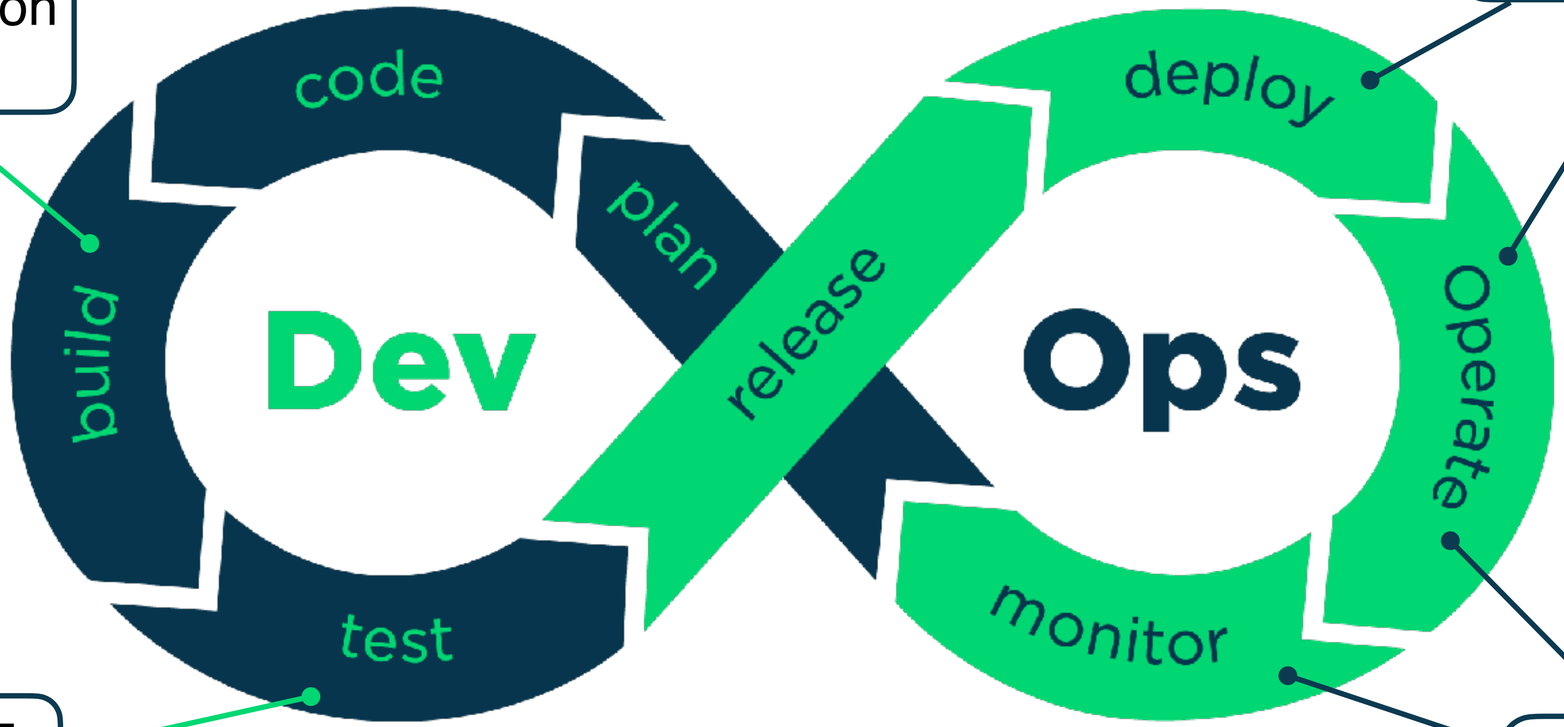
On 29th of September 2018, it was discovered that the **credentials for the cloud storage** of an indoor **camera** were **hardcoded** in its software.

With those credentials, anyone could access the uploaded videos from any customer.

Security throughout the SDLC

SCA
Software Composition Analysis

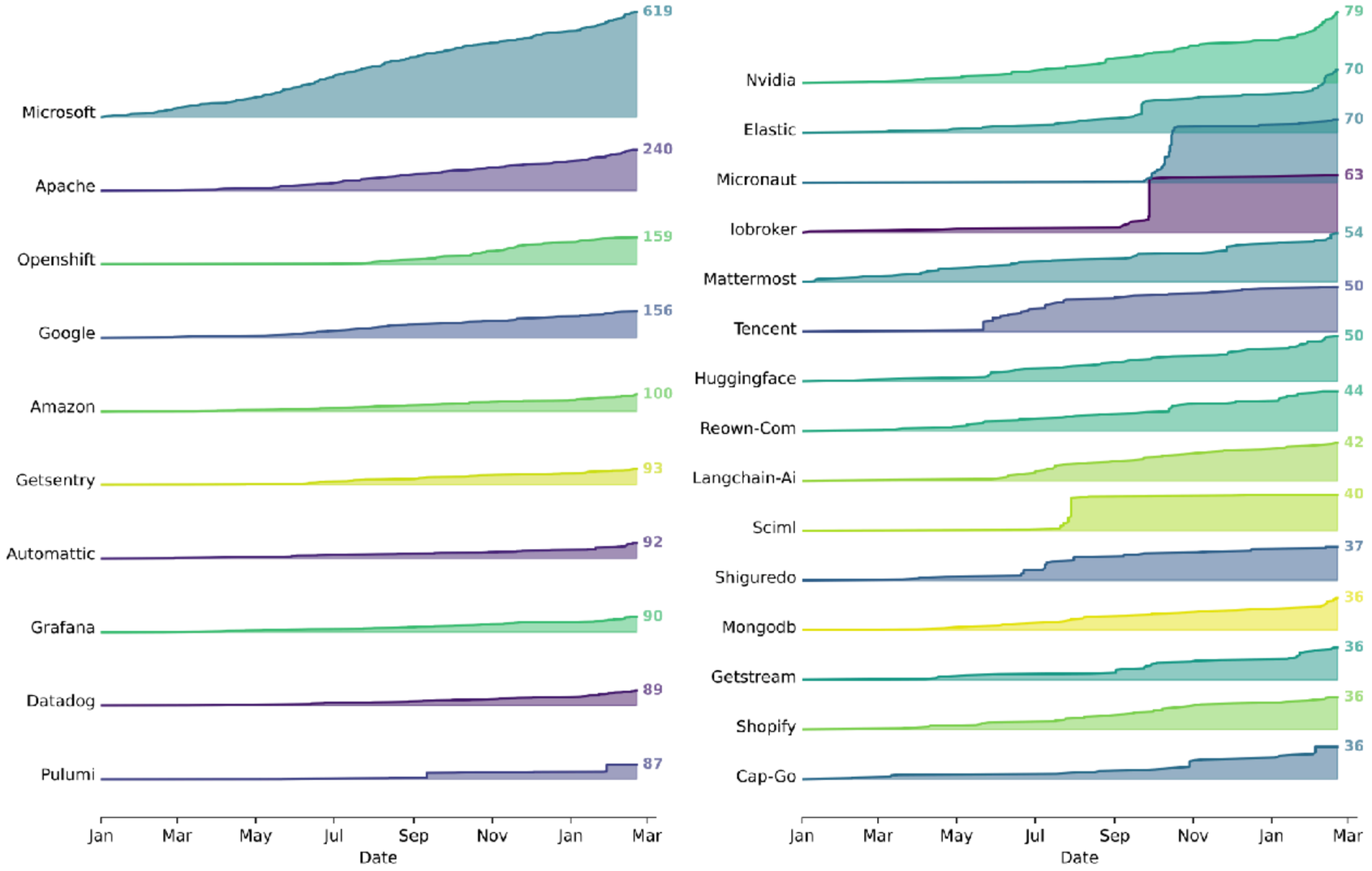
IaC
SAST & SCA for Ansible, Docker, …



SAST & DAST
Application Security Testing

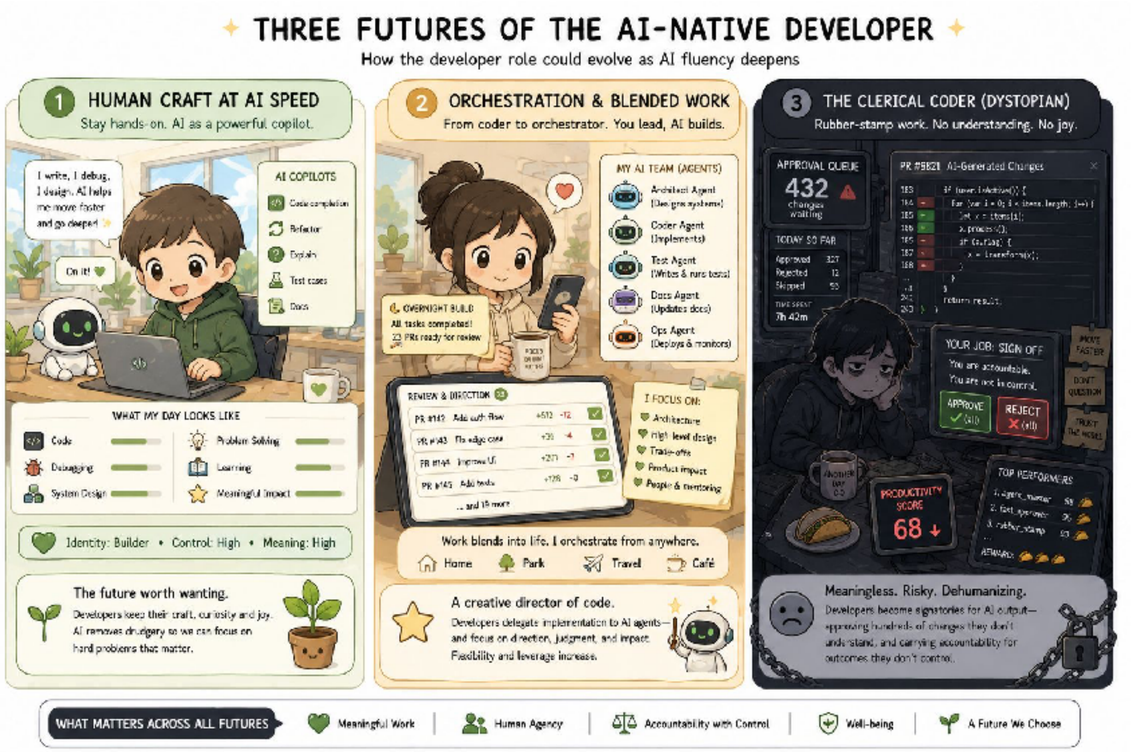
PaC
SIEM

But remember … times are changing …

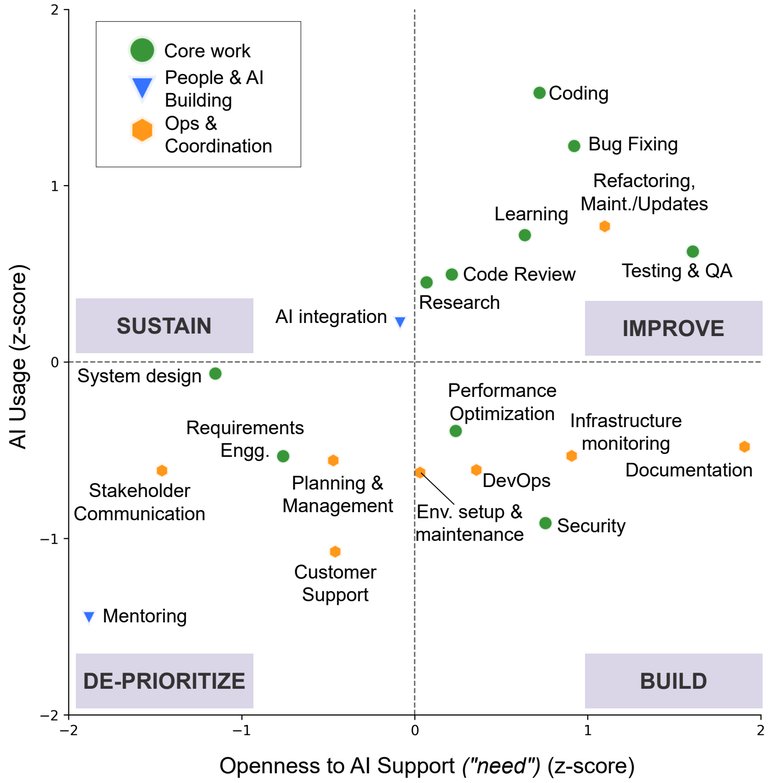


[3] Robbes, R., Matricon, T., Degueule, T., Hora, A., & Zacchiroli, S. (2026). Agentic Much? Adoption of Coding Agents on GitHub. *arXiv preprint arXiv:2601.18341*.

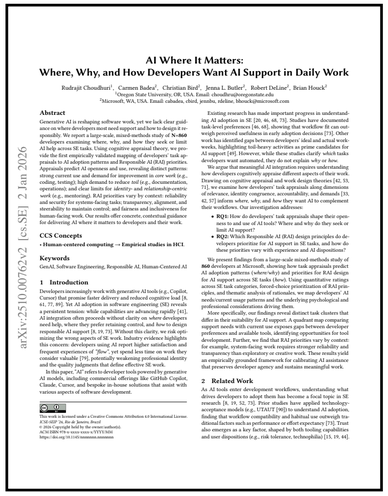
… and will continue to change!



https://queue.acm.org/detail.cfm?id=3807961



https://cabird.github.io/AI_where_it_matters/



AI Where It Matters:
Where, Why, and How Developers Want AI Support in Daily Work

Rudrajit Choudhuri[1], Carmen Badea[2], Christian Bird[2], Jenna L. Butler[2], Robert DeLine[2], Brian Houck[2]
[1]Oregon State University, OR, USA. Email: choudhur@oregonstate.edu
[2]Microsoft, WA, USA. Email: cabadea, chird, jenabu, rdeline, bhouck@microsoft.com

arXiv:2510.00762v2 [cs.SE] 2 Jan 2026

[Choudhori et al. 2026]

from a survey of 1193 Microsoft developers

Let's **collaborate** on tools and processes to realise possible future 1 & 2!

Example: how to choose the LLM for our agents?

Popularity: “Everyone is using it”.

Capability: “It is a code-LLM, it works”.

Benchmarks: “It’s the best on BigCodeBench”.

Limitations of LLM benchmarks

LLM-generated method

```
char *content = read_file_from_dir(argv[1], argv[2]);
```

Most benchmarks measure the capability of the LLM in generating functionally correct code through pre-determined tests.

but functional correctness ≠ security correctness

functional correctness:
does what it's supposed to do

security correctness:
does not do what it's **not** supposed to

Passes functional test: read_file_from_dir("notes.txt") returns content

Security vulnerability: Attacker passes ../../etc/passwd (path traversal, CWE-22)

Smarter, but Not Safer: An Empirical Analysis of the Functional-Security Gap in Evolving LLMs

Alfonso Cannavale*, Gilberto Recupito†, Coen De Roover†, Fabio Palomba*, Andrea De Lucia*
*Software Engineering Salerno (SeSa) Lab, University of Salerno, Fisciano, Italy
Email: acannavale@unisa.it, fpalomba@unisa.it, adelucia@unisa.it
†Software Languages Lab, Vrije Universiteit Brussel, Brussel, Belgium
Emails: gilberto.recupito@vub.be, coen.de.roover@vub.be

***Abstract*—Large Language Models (LLMs) have become tools for software development and maintenance, demonstrating impressive capabilities in automating tasks such as code generation and refactoring. However, it remains empirically unclear whether the rapid scaling of these capabilities is accompanied by proportional improvements in security, or whether a discrepancy, defined as the Security–Functionality Gap, persists across model generations. To address this question, we conducted an evolutionary study analyzing the trajectory of 12 state-of-the-art models across the GPT, Llama, and Claude Sonnet lineages. By leveraging an outcome-driven benchmark based on dynamic execution, we quantified the divergence between the models' ability to generate functionally correct code versus secure code over four sequential versions per lineage. Results reveal that, while functional correctness has effectively plateaued (> 90%) across all families, functional-security correctness has evolved more slowly. Specifically, while proprietary models demonstrate marginal improvements, open-weight models exhibit stagnation, and the most advanced models occasionally introduce new regressions due to increased code complexity. These findings suggest that security alignment is not an intrinsic byproduct of model scaling, highlighting the urgent need for external guardrails in AI-assisted maintenance pipelines.**

***Index Terms*—Large Language Models, Software Evolution, Software Security, Empirical Software Engineering.**

I. Introduction

Large Language Models (LLMs) have rapidly transitioned from experimental tools to fundamental components of the software development lifecycle. They are now pervasive in critical software maintenance tasks, including automated refactoring, program repair, and code optimization [1]. Modern Integrated Development Environments (IDEs) increasingly embed these models directly into the workflow, enabling developers to delegate complex reasoning and code modification tasks to AI agents [2]. In selecting these agents, practitioners heavily while benchmarks show exceptional performance when using LLMs, recent studies confirmed the criticality of using LLM-generated code, with risk of introducing unsafe code during the development process [5], [6].

Among quality trade-offs, security represents one of the most critical concerns. Early empirical studies on commercial LLMs demonstrate that, despite their apparent proficiency, these models frequently generate insecure and vulnerable code [7], [8]. Following the first wave of ChatGPT adoption, subsequent evidence showed that insecure code suggestions were often incorporated into production systems via pull requests, partly due to developers' overtrust in model outputs [9].

While these studies provide crucial insights, they share a common limitation: they primarily adopt a *cross-sectional perspective*, evaluating different models or a single model version at a specific point in time. Such evaluations are valuable for assessing the immediate risk associated with a specific model version. However, by design, cross-sectional analyses do not consider temporal dynamics, hence failing to capture how model properties evolve over successive releases. As a result, they lack an *evolutionary perspective* [10], which is essential for understanding whether improvements in functional capabilities are accompanied by sustained, stagnant, or even regressive security guarantees over time. Such insights may contribute to the current body of knowledge by enabling a deeper understanding not only of how well LLMs perform at a given time, but also of how their functional and security properties co-evolve over time.

To address this gap, we introduce the notion of the **Security-Functionality Gap** (Δ_{Sec}). While its mathematical formulation is detailed in Section IV, we informally define this gap as the quantitative divergence between a model's *Functional Cor-*

How large is the security-functionality gap in LLMs?

This is the security-functionality gap ($\Delta_{Sec}@k$).

$$func@k = \mathbb{E}\left[1 - \frac{\binom{n - c_{func}}{k}}{\binom{n}{k}}\right]$$

Given the set of LLM solutions that passes at K attempts.

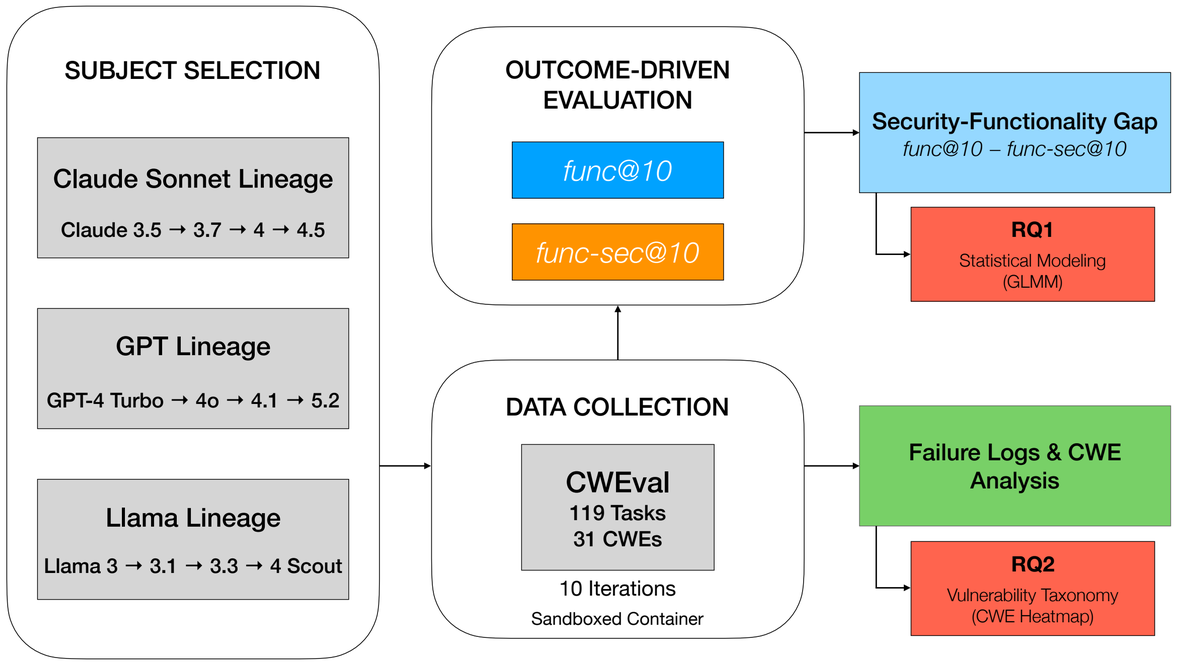
$$func\_sec@k = \mathbb{E}\left[1 - \frac{\binom{n - c_{func\_sec}}{k}}{\binom{n}{k}}\right]$$

Given the set of LLM solutions that passes at K attempts **and produce secure code**.

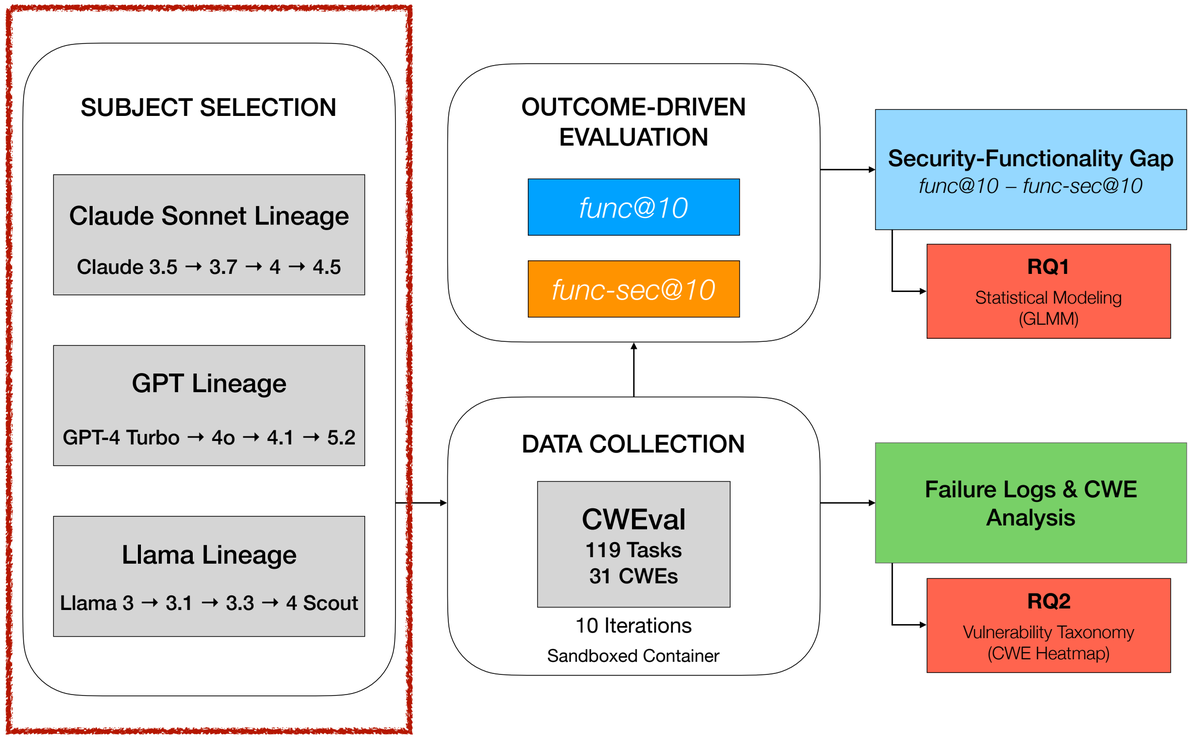
$$\Delta_{Sec}@k = func@k - func\_sec@k$$

Defined as the gap between code that passes functional tests and code that passes both functional AND security tests.

An Empirical Analysis of the Security-Functionality Gap in Evolving LLMs

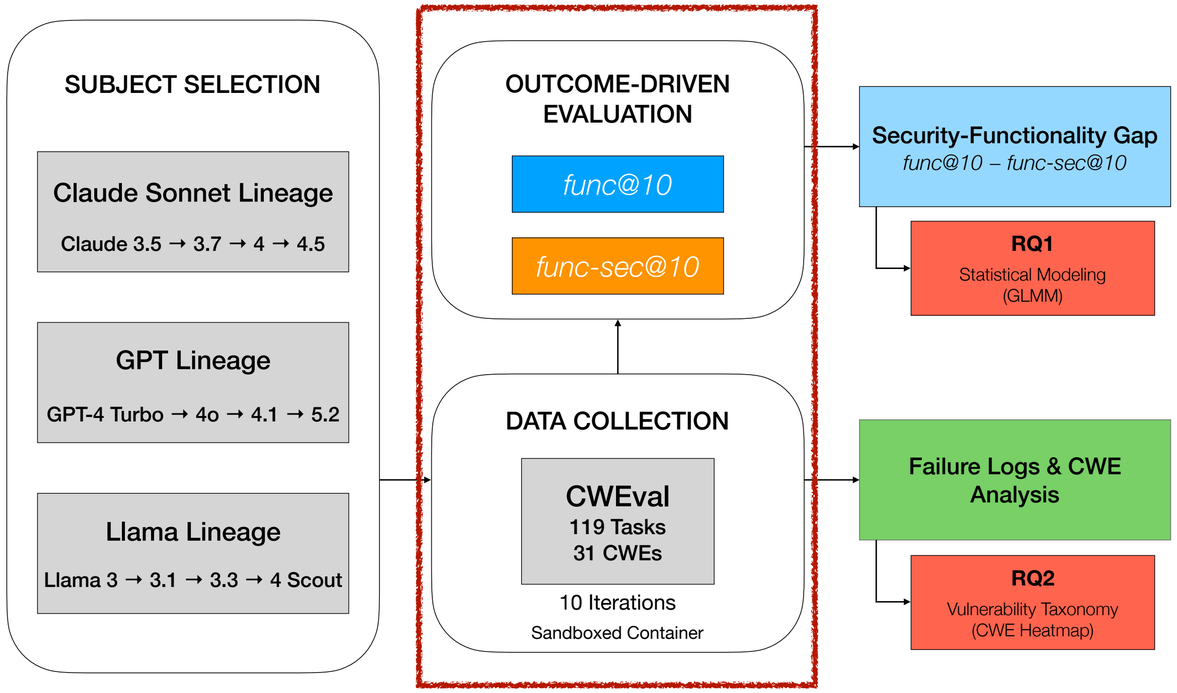


An Empirical Analysis of the Security-Functionality Gap in Evolving LLMs



We selected 3 model families and interacted with each subsequent model generation, until the latest generation.

An Empirical Analysis of the Security-Functionality Gap in Evolving LLMs



We used **CWEval benchmark**, containing both functional and security tests.

LLM-generated method

```
char *content = read_file_from_dir(argv[1], argv[2]);
```

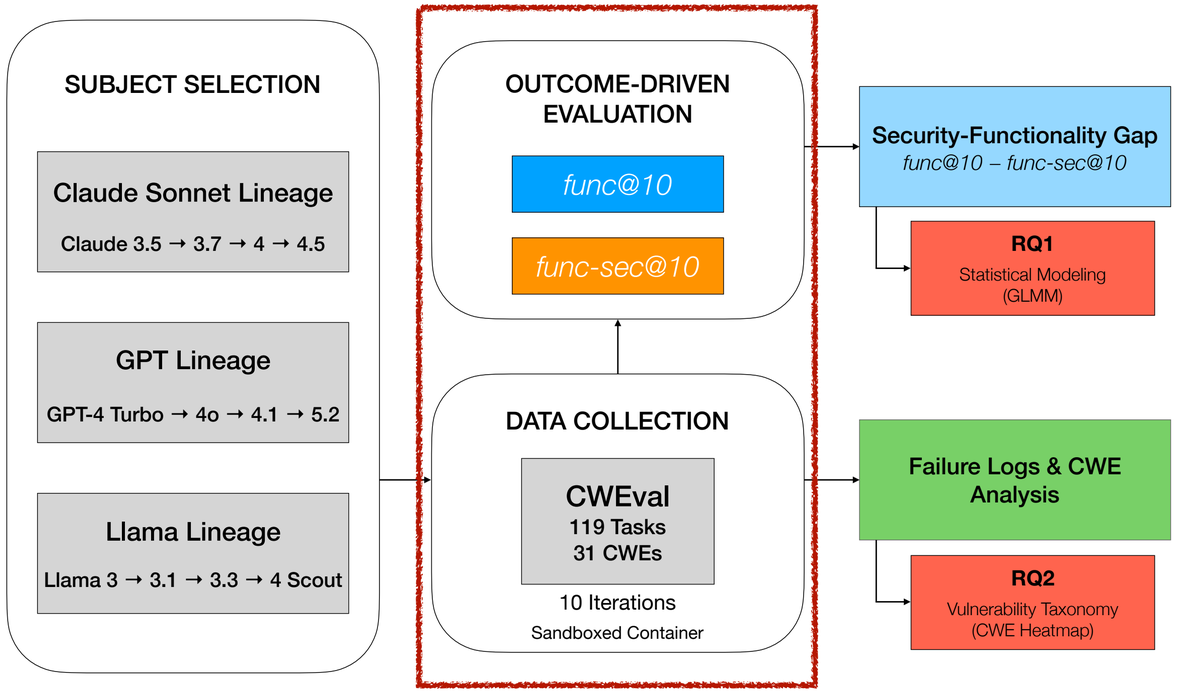
Functional test

```
pytest.param({'a.txt': 'a', './b.txt': 'b'}
```

Path Traversal Security test (CWE-22)

```
pytest.param('../../z.txt: 'txt z')
```

An Empirical Analysis of the Security-Functionality Gap in Evolving LLMs



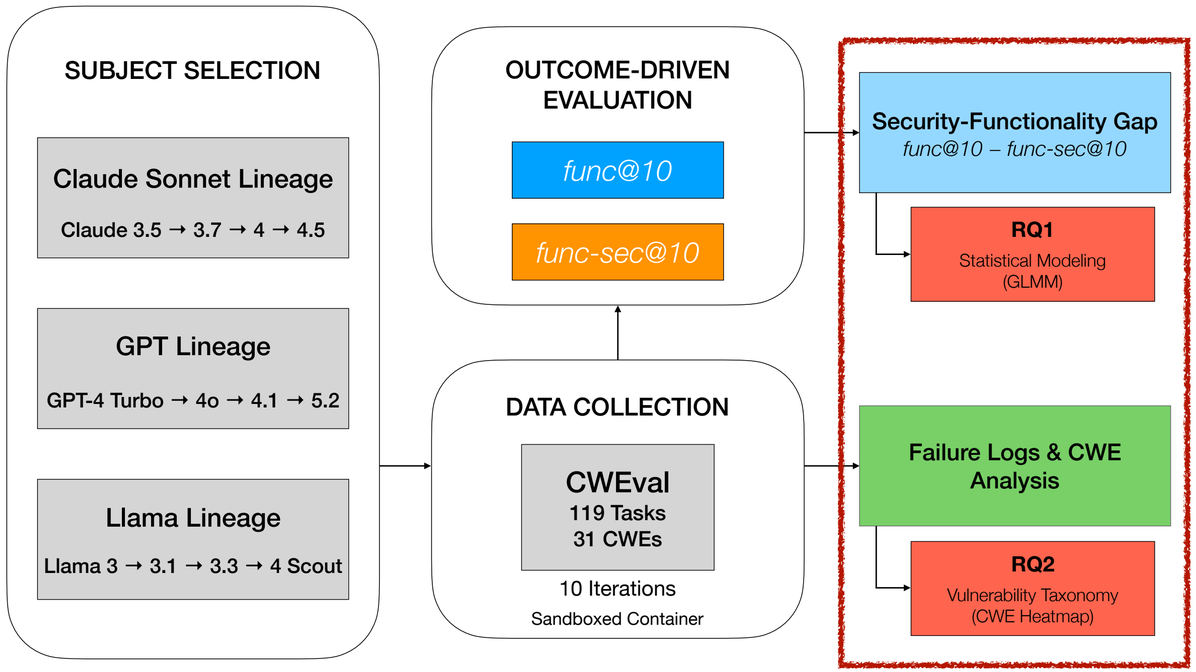
We used **CWEval benchmark**, containing both functional and security tests.

119 security-critical code tasks

31 CWE-specific tests

5 programming languages

An Empirical Analysis of the Security-Functionality Gap in Evolving LLMs



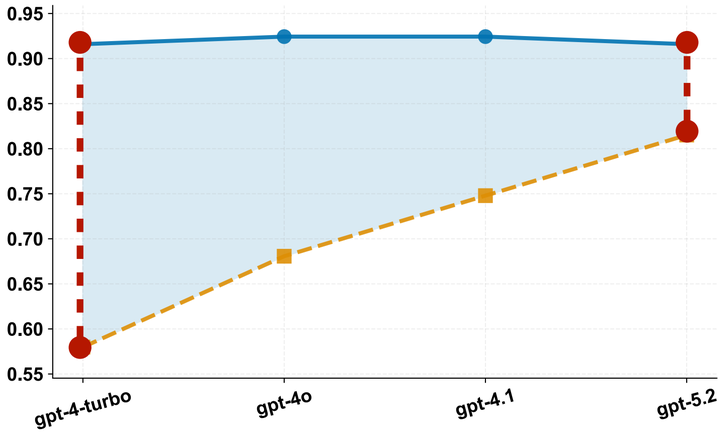
RQ1: *How does the Security-Functionality Gap evolve across successive generations of LLMs?*

RQ2: *What vulnerability classes characterize the Security-Functionality Gap throughout LLM evolution?*

Finding 1: Functional correctness is high, security not

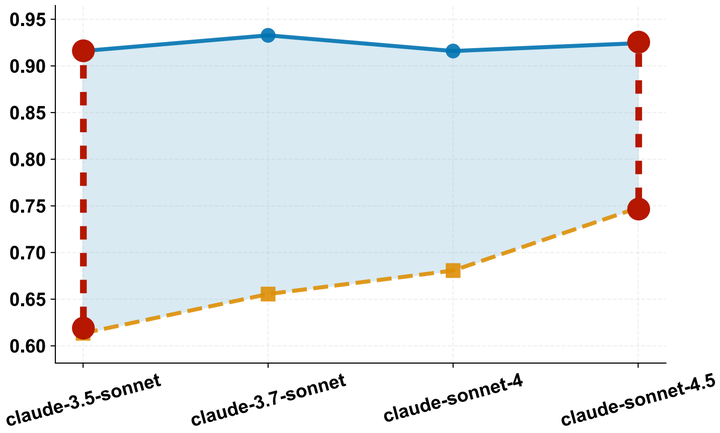
: func@k : $\Delta_{Sec}@k$: func_sec@k

GPT-Family



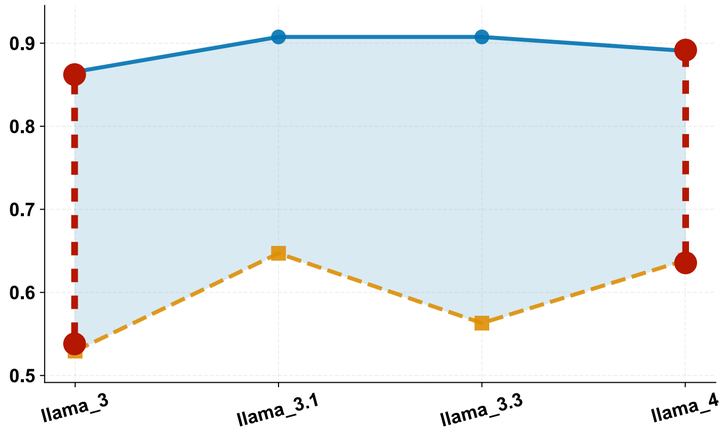
:35-10%

Claude Sonnet Family



:30-18%

Llama Family



:35-25%

That means:

GPT:
1 in 10 functionally correct solutions is not secure

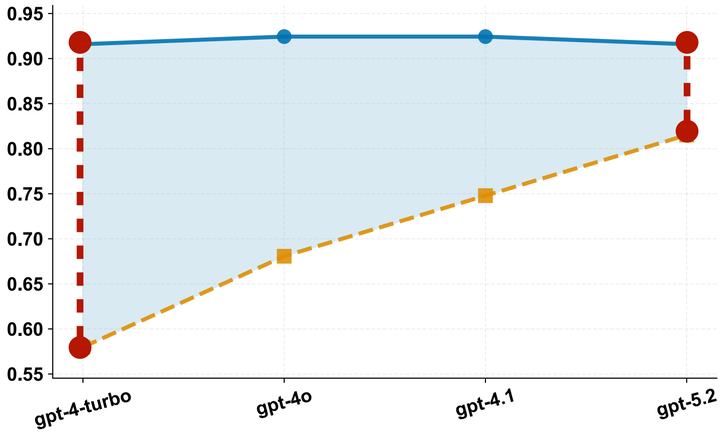
Claude Code:
2 in 10 functionally correct solutions is not secure

Llama:
4 in 10 functionally correct soultions is not secure

Finding 1: Functional correctness is high, security not

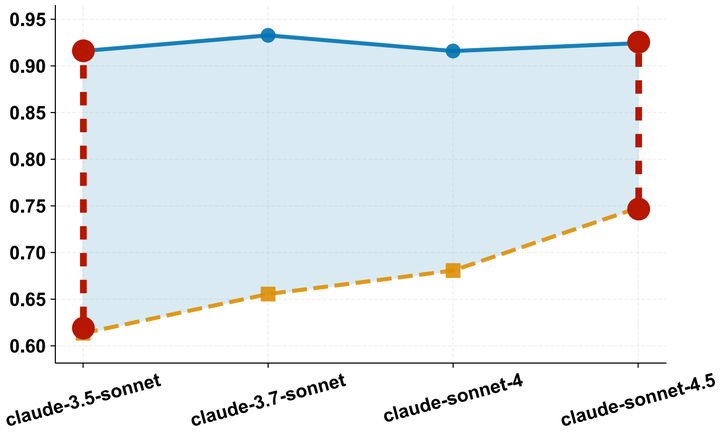
: func@k : $\Delta_{Sec}@k$: func_sec@k

GPT-Family



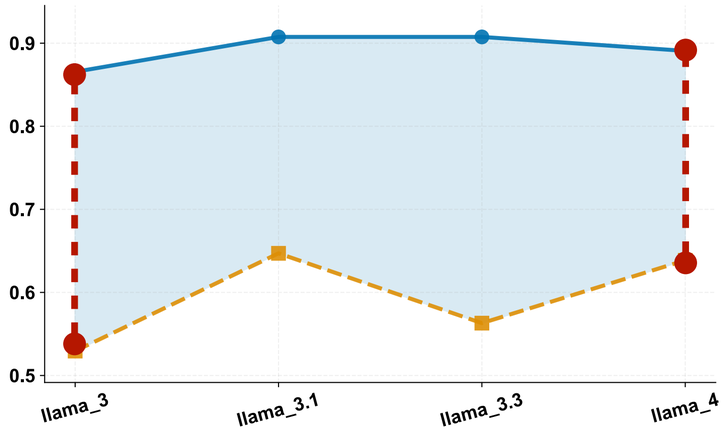
:35-10%

Claude Sonnet Family



:30-18%

Llama Family

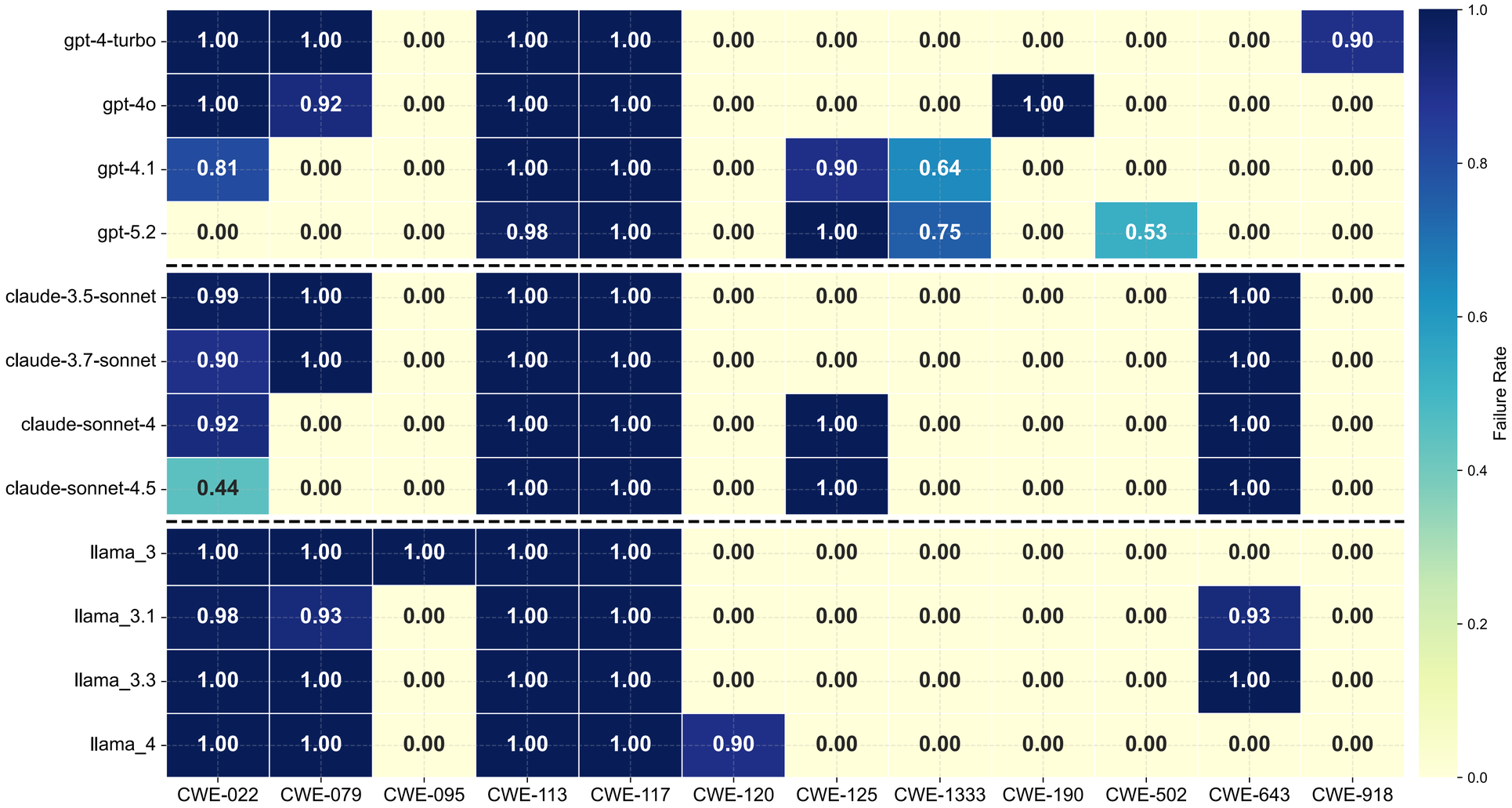


:35-25%

Secure code generation is NOT automatically learned by newer models

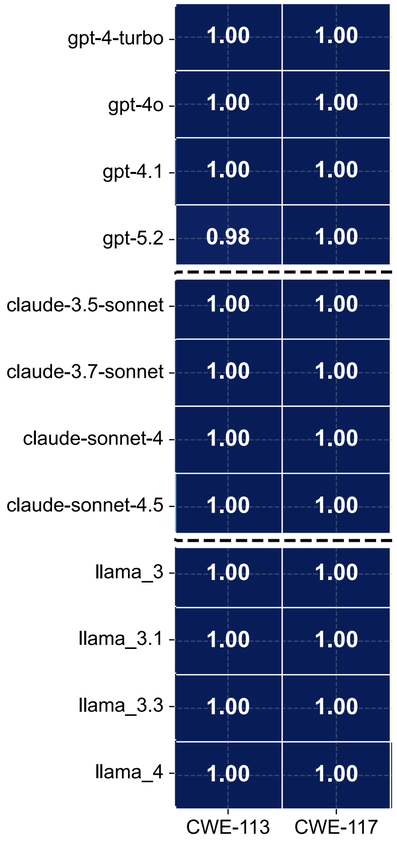
While frontier models improved the security of their generated code, they are not as secure as they are functional.

Finding 2: Over time, we found recurrent patterns of vulnerability proneness among model versions.



Finding 2: Over time, we found recurrent patterns of vulnerability proneness among model versions.

Pattern 1: Persistent Blind Spots. This category includes vulnerabilities that are still unaddressed by LLMs over all model evolutions.



CWE-117 (Log Injection)

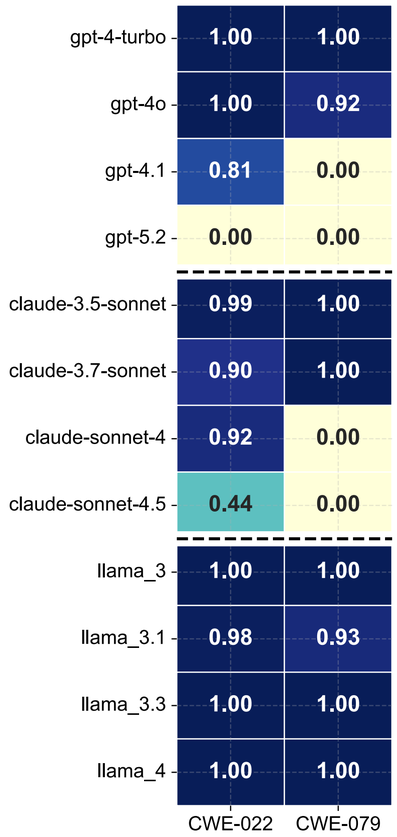
Security Test Result for Claude-Sonnet 4.5:
Injected `'Hello, World!\n[{CUR_TIMESTAMP}] Received: User admin logged in.'` to insert a fake log entry.

CWE-113 (HTTP Response Splitting)

Security Test Result for Llama 4:
Injected `'en-US\nHTTP/1.1 200 OK'` inside HTTP request.

Finding 2: Over time, we found recurrent patterns of vulnerability proneness among model versions.

Pattern 2: Evolutionary Divergence. This category denotes different behaviours between open source and proprietary lineages.



CWE-22 (Path Traversal)

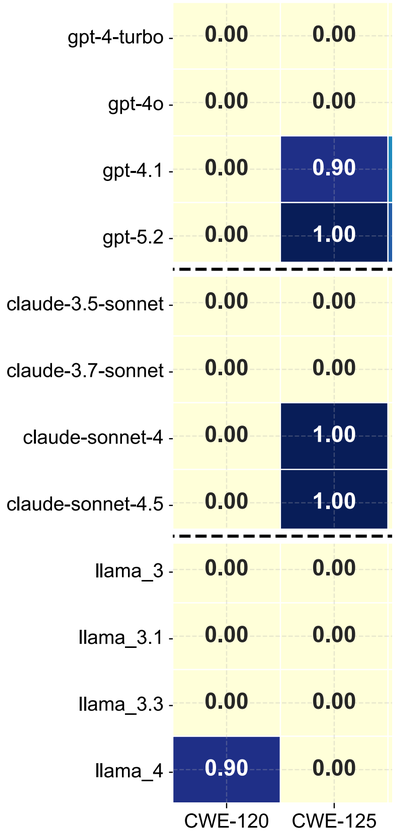
Security Test Result for Llama 4:
Injected `'./../../cs.txt'` to read restricted files.

CWE-79 (Cross-site Scripting)

Security Test Result for Llama 4:
Injected `'<script>alert("XSS")</script>'` inside the web script.

Finding 2: Over time, we found recurrent patterns of vulnerability proneness among model versions.

Pattern 3: Regression Anomalies. This category denotes vulnerabilities exclusively introduced by newer models.



CWE-120 (Classic Buffer Overflow) - Security test exclusive for C

Security Test Result for Llama 4:
Injected `'a' * 31` into 30 character buffer size.

CWE-125 (Out of Bound Read) - Security test exclusive for C

Security Test Result for Claude-Sonnet 4.5:
Accessed to memory with index `-1` through the allocated array.

If you do not care about the security of your code, the LLM won’t either.

Practical suggestions

1

Do Not Evaluate Coding Models Only with Functional Tests

Functional tests answer: Does the code work?

Security tests answer: Can the code be abused?

Practical suggestions

1 Do Not Evaluate Coding Models Only with Functional Tests

Functional tests answer: Does the code work?

Security tests answer: Can the code be abused?

2 Treat Model Upgrades as Security-Relevant Changes

Ask: Does the new model introduce fewer vulnerabilities?

Compare: Is the security pass rate lower than before?

Inspect: Let’s do regression tests. Is the model introducing new vulnerabilities that weren’t there before?

Test: Use the LLM for different controlled and testing tasks before relying on it.

Practical suggestions

1 Do Not Evaluate Coding Models Only with Functional Tests

Functional tests answer: Does the code work?

Security tests answer: Can the code be abused?

2 Treat Model Upgrades as Security-Relevant Changes

Ask: Does the new model introduce fewer vulnerabilities?

Compare: Is the security pass rate lower than before?

Inspect: Let's do regression tests. Is the model introducing new vulnerabilities that weren't there before?

Test: Use the LLM for different controlled and testing tasks before relying on it.

3 Make Security Requirements Explicit in Prompts

Bad prompt:

"Write a function to read a file by name"

Good prompt:

"Write a function to read a file by name. **Prevent path traversal, validate the filename, restrict access to allowed directory, reject malicious input**"

Limitations of LLM benchmarks

Most benchmarks measure the capability of the LLM in generating functionally correct code through pre-determined tests.

LLM-generated method

```
char *content = read_file_from_dir(argv[1], argv[2]);
```

but functional correctness ≠ security correctness

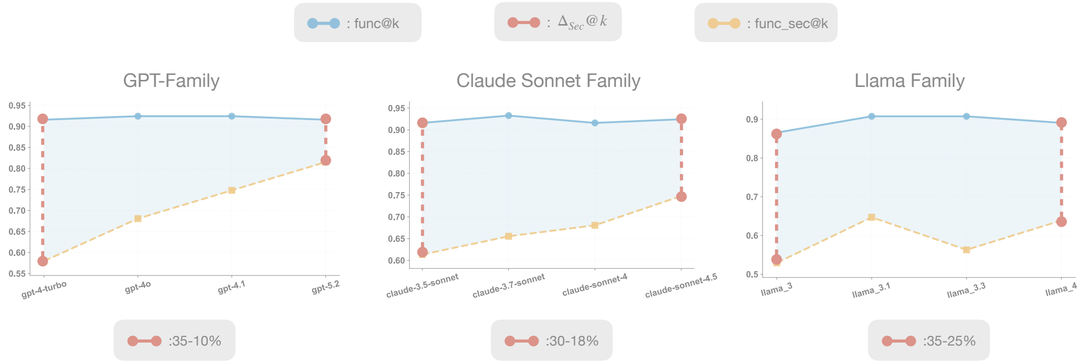
functional correctness:
does what it's supposed to do

Passes functional test: read_file_from_dir("notes.txt") returns content

security correctness:
does not do what it's **not** supposed to

Security vulnerability: Attacker passes ../../etc/passwd (path traversal, CWE-22)

Finding 1: Functional correctness is high, security not



That means:

GPT:	Claude Code:	Llama:
1 in 10 functionally correct solutions is not secure	2 in 10 functionally correct solutions is not secure	4 in 10 functionally correct soultions is not secure

Q&A

Finding 2: Over time, we found recurrent patterns of vulnerability proneness among model versions.

Pattern 1: Persistent Blind Spots. This category includes vulnerabilities that are still unaddressed by LLMs over all model evolutions.

CWE-117 (Log Injection)

Security Test Result for Claude-Sonnet 4.5:
Injected `'Hello, World!\n[{CUR_TIMESTAMP}] Received: User admin logged in.'` to insert a fake log entry.

CWE-113 (HTTP Response Splitting)

Security Test Result for Llama 4:
Injected `'en-US\nHTTP/1.1 200 OK'` inside HTTP request.

Practical suggestions

1 **Do Not Evaluate Coding Models Only with Functional Tests**

Functional tests answer: Does the code work?

Security tests answer: Can the code be abused?

2 **Treat Model Upgrades as Security-Relevant Changes**

Ask: Does the new model introduce fewer vulnerabilities?

Compare: Is the security pass rate lower than before?

Inspect: Let's do regression tests. Is the model introducing new vulnerabilities that weren't there before?

Test: Use the LLM for different controlled and testing tasks before relying on it.

3 **Make Security Requirements Explicit in Prompts**

Bad prompt:
"Write a function to read a file by name"

Good prompt:
"Write a function to read a file by name. **Prevent path traversal, validate the filename, restrict access to allowed directory, reject malicious input**"

Security Quality: Findings From Our Empirical Studies

Coen De Roover, Gilberto Recupito. Vrije Universiteit Brussel.