A Secure Mocking Approach towards Software Supply Chain Security

Daisuke Yamaguchi, Shinobu Saito, Takuya Iwatsuka, Nariyoshi Chida
NTT, Inc., Tokyo, Japan
{daisuke.yamaguchi, shinobu.saito, takuya.iwatsuka, na.chida}@ntt.com

Tachio Terauchi
Waseda University, Tokyo, Japan
terauchi@waseda.jp

***Abstract*—As software development increasingly relies on external collaboration, organizations face new risks of intellectual property leakage beyond traditional concerns about deployed software. Even when the source code is protected, adversaries may infer sensitive internal program specifications by observing the program behavior during the development and testing phases.**

This paper addresses the problem of specification leakage through behavioral observation in collaborative software development. We propose a novel software development method that centers on specially crafted test doubles referred to as *secure mocks*. Secure mocks serve as drop-in replacements for original components during development and testing while preventing the exposure of sensitive internal specifications through observable behavior. We formalize the correctness conditions for secure mocks and define the secure mock construction problem as a constraint satisfaction problem parameterized by the program to protect, the development specification, and a security policy. Our approach enables secure test-driven development (TDD) with external collaborators, bridging the gap between traditional TDD styles. We discuss the implications for secure collaboration with external developers and outline future research directions for automating secure mock generation and integrating this paradigm into real-world development pipelines.

***Index Terms*—Secure mock, TDD, information leakage**

I. Introduction

For organizations, protecting the internal specifications of programs has become a critical concern. Particularly, when these specifications are considered the foundation of their intellectual property and competitive edge, they must be prevented from being leaked to external parties. Traditionally, obfuscation and encryption are common techniques for protecting completed software artifacts from illegal copying, tampering, and reverse engineering by malicious users [1], [2].

The landscape of software development has evolved with the widespread adoption of external collaboration [3]–[5]. Recent online remote repositories provide platforms for sharing not only the code but also the development environments with external contributors (e.g., GitHub Codespaces [6]). Crowdsourcing parts of the development to anonymous crowd workers has also become a common practice [7]–[9].

These trends expand the scope of potential information leakage beyond just the deployed software to the software development life cycle itself. Particularly, the external vendors and developers in software supply chain are increasingly recognized as an attack vector for potential leaks even in the traditional outsourced development [10], [11]. Several techniques and tools have been proposed to monitor and detect source code leaks from public repositories [12]–[15]. For proactive approaches, MORDEn [16] provides capabilities to prevent source code leakage in collaborative development environments. Nevertheless, even if the source code is protected, there remains a significant risk: adversaries may infer internal program specifications by observing the program's behavior (e.g., [17]–[21]) during development and testing.

In this paper, we address the problem of internal specification leakage via behavioral observation in the development process. We propose a software development methodology that leverages a specially crafted test double, called a *secure mock*. A secure mock is designed so that it can be used as a drop-in replacement for the original component during development and testing, while ensuring that sensitive internal specification is not exposed through its observable behavior. A secure mock provides capabilities to securely work test-driven development (TDD) [22], [23] collaborating with external developers. We refer to this style of TDD as the *Hachioji school* and aim to hit the sweet spot between the London school (mockist) [22] which favors aggressive abstraction and mocking, and the Detroit school (classicist) [23] which favors using real components; by enabling practical development and testing workflows while mitigating the risk of information leakage.[1]

The remainder of this paper is organized as follows. In Section II, we present a motivating example to illustrate the information leakage problem and introduce the concept of secure mock and the Hachioji school. Section III formalizes the problem setting, defines the requirements for secure mocks, and frames the secure mock construction problem, which is configured by a program to protect, a specification of a program to develop, and a security policy that defines the information leakage constraints. Section IV reviews related work in data protection and foundations of modeling and bounding information leakage. Finally, Section V concludes the paper and discusses directions for future research.

II. Illustrative Example

Let us consider a scenario in which a health insurance company develops an internal system that calculates and displays insurance premiums based on customer information input. The business logic for the premium calculation is developed in-

[1]Hachioji is a town in Japan geographically located at about the mid-point between London and Detroit.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00352

house, while the user interface, which receives customer information and displays the information, the calculated premium, and discount offers, is outsourced to reduce the development costs. The requirements for the outsourced user interface are *(R0) forms to input customer name, age, and residence are provided, (R1) the entered customer information is sent to the premium calculation service, (R2) the premium is sent to the discount offering service, which returns available discount offers for the premium, and (R3) the customer information, the premium calculated by the premium calculation service, and the discount offers returned by the discount offering service are displayed on the view.* Acceptance criteria require that tests confirming the correct implementation of requirements R0, R1, R2, and R3 are passed.

A. Two Schools of Testing and Information Leakage

To ensure that the outsourced implementation meets the acceptance criteria, it is necessary to provide test programs in advance. There are two main schools of thought regarding the construction of such test programs: the London school (mockist) [22] and the Detroit school (classicist) [23]. The London school is characterized by actively mocking components that the system under test depends on, making it suitable for verifying business logic. In contrast, the Detroit school is reluctant to use mock dependencies, instead using actual components in the test program, which is preferable when verifying input/output, such as user interfaces.

To verify requirements R0, R1, R2, and R3, it is desirable to provide a Detroit-style test program, as it is well-suited for input/output verification. However, creating a Detroit-style test program requires access to the actual premium calculation service. This service often contains proprietary risk assessment knowledge developed by the insurance company, and providing it to the outsourcing vendor poses a risk of intellectual property leakage.

For instance, consider the premium calculation service that calculates the premium by customer age and residence, as Table I shows. The premium calculation service reflects that *(a)* Residence *is a key parameter to the calculation, and (b) the* Buzz City *residents are applied a special pricing strategy that is different from other residents*. These characteristics can potentially be inferred by observing the input/output values of the premium calculation service. If (a) and (b) represent the insurance company's proprietary risk assessment knowledge, providing the actual premium calculation service to the outsourcing vendor poses a risk of intellectual property leakage, even if its source code is hidden.

TABLE I
PREMIUM CALCULATION COMPONENT

a:**Age**	**Residence**	**Premium**
$a \geq 0$	Foo Town	1,000
$0 \leq a \leq 50$	Buzz City	1,000
$a > 50$	Buzz City	1,500
$a \geq 0$	Barpolis	1,000

B. The Hachioji School and Secure Mock

The *Hachioji school* is a testing approach that aims to minimize the risk of confidential information leakage to outsourcing vendors in the Detroit-style testing under untrusted development environments. In the Hachioji-style testing, a mock of the premium calculation component is used. We refer to the mock as a *secure mock*, which is a mock of the original dependency that is constructed in such a way that it can be used in place of the original component in the test program while preventing the leakage of intellectual property.

Consequently, the main interests of the Hachioji school and secure mock are twofold:

Security policy on Secure Mock is conditions on a secure mock to protect intellectual properties such as (a) and (b) in the premium calculation service example. For the intellectual properties (a) and (b), we can define conditions for information protection by the notion of information flow [24]. Namely, the security policy on a secure mock can be defined as *noninterference* (or *no information leakage*) [25], which means that the input values of a sensitive parameter have no effect on output values. By mapping the key parameter *Residence* to the sensitive parameter, the intellectual properties (a) and (b) can be protected. It is also an interesting problem to consider possible weakening of the conditions for individual intellectual properties.

Secure Mock Construction concerns methods to create a secure mock from the original dependency. In the premium calculation service example, replacing the residence values with dummy values to ensure noninterference seems to be a solution, but this idea makes it impossible to confirm the acceptance criteria, which is to verify the interface functionality, such as providing input forms (R0) and displaying actual residence names (R3). Similarly, if we replace the premium value with a dummy value, we would not be able to verify the functionality of the discount offering service (R2). A secure mock ensures that no confidential information leakage occurs while allowing tests equivalent to those using the original dependency. In the next section, we formalize these problem settings and the properties required for a secure mock.

III. PROBLEM SETTING

In this section, we first formalize the problem settings of outsourced software development. Then, we discuss the properties required for a secure mock and define its construction problem.

A. Outsourced Development and Security Properties

Definition 1 (Notational Conventions)**.** Let G be a program component under development and F be a program component that G depends on. Let γ be a specification of G. We write $F \triangleright G :: \gamma$ to denote that the component G satisfies γ under a component F.

Since multiple program components can be aggregated into a single component (e.g., take the Cartesian product of the components), we assume that G depends on a single program component F for simplicity. The specification γ is often expressed in a formal language or as test cases.

Definition 2 (Outsourced Development)**.** An outsourced development is a process where external software developers implement a program component G such that $F \triangleright G :: \gamma$.

When the external software developers are in another country or crowd workers, the outsourced development is respectively called offshoring [26] or crowdsourcing [9]. As a way to implement a program component G which satisfies γ, test-driven development (TDD) is a well-known approach in software development practice, including outsourced development. TDD in outsourced development lets the external software developers take the program component F and the specification γ as test cases from the project owners to implement the program component G.

As remarked before, because the component F often contains an intellectual property of the project owner, its implementation should be hidden from external software developers by obfuscating [2] the source programs or providing only a service endpoint [16]. However, even if F's implementation is hidden, the risk of intellectual property leakage still exists. External software developers may be possible to infer F's intellectual property by observing its input/output behavior. To model a security policy against this type of information leakage, non-interference seems to be a promising formalism.

Definition 3 (Noninterference [25])**.** A program F with parameters X and $\bar{X}$ is *noninterferent* with respect to X iff the following proposition $\Psi(F, X)$ is true.

$$\Psi(F, X) := \forall x_1, x_2 \in X, \forall \bar{x} \in \bar{X}. F(x_1, \bar{x}) = F(x_2, \bar{x})$$

In the above definition, the parameters x and $\bar{x}$ are referred to as sensitive and non-sensitive parameters, respectively. Note that x and $\bar{x}$ can be a tuple so that the above covers the case where multiple parameters can be sensitive or non-sensitive.

Example 1. Let F be the premium calculation component of Table I. $\Psi(F, \text{Residence})$ is false because $F(\text{Foo Town}, 60) = 1000 \neq 1500 = F(\text{Buzz City}, 60)$.

Example 1 indicates that external software developers may infer the intellectual properties (a) and (b) of Section II-A.

Since noninterference prohibits any information leakage, it is the strongest policy that can be enforced. It is possible to weaken the security policy depending on the intellectual properties to be protected or an acceptable amount of leakage [27]. For instance, consider the case where the intellectual property to be protected is just (b) instead of (a) and (b). The predicate Ψ can be weakened to the predicate $\forall x_1, x_2 \in X, \forall \bar{x} \in \bar{X}. x_1 = \text{Buzz City} \Rightarrow F(x_1, \bar{x}) = F(x_2, \bar{x})$.

B. Secure Mock and Its Construction

Next, we model outsourced software development under a security policy and formalize the notion of secure mock.

Consider the outsourced development defined in Definition 2 where the program F does not satisfy a security policy Ψ. In the Hachioji school of TDD that we propose, we provide a test double $\mathcal{S}(F)$, where $\mathcal{S}$ is a program transformation that transforms the given component to a test double. We state the properties that are desired for $\mathcal{S}(F)$. First, $\mathcal{S}(F)$ should be secure and does not leak proprietary information, that is, it should satisfy $\Psi(\mathcal{S}(F), X)$ with X being the sensitive parameter. We denote this by $\triangleright\mathcal{S}(F) :: \Psi$. Second, the external developers, receiving $\mathcal{S}(F)$ and a possibly *modified* specification $\Theta(\gamma)$ where γ is the original specification of the target program component G, should implement G' satisfying $\mathcal{S}(F) \triangleright G' :: \Theta(\gamma)$. We describe the specification modification next.

Since $\mathcal{S}(F) \triangleright G' :: \gamma$ does not necessarily imply $F \triangleright G' :: \gamma$, after receiving the deliverable G' from the external developers, the project owner performs rework to ensure that $F \triangleright G' :: \gamma$. There may be many ways to design such rework, as well as ways to design the secure mock $\mathcal{S}(F)$. One way that we propose in this paper is to modify the specification for the external software developers. Let Θ be a modification of the specification γ which is a mapping between specifications denoted by $\Theta : \mathsf{Spec} \rightarrow \mathsf{Spec}$ where Spec is the set of specifications. The external software developers' specification is denoted by $\Theta(\gamma)$. The specification modification restricts the externally developed program G' to satisfy a certain "good" property that is not necessarily the same as γ but one that allows us to rework G' into a component that satisfies γ when given the actual program component F.

Here, $\mathcal{R}$ is a mapping between programs. The procedure of outsourced development with secure mock is summarized in the diagram shown in Fig. 1, where the dashed arrow indicates objective development using actual program F and the solid arrows indicate outsourced development procedures using test double.

$$\begin{array}{ccc} F & \dashrightarrow & F \triangleright \mathcal{R}(G') :: \gamma \\ {\scriptstyle\mathcal{S}}\downarrow & & \uparrow{\scriptstyle\mathcal{R}} \\ \triangleright\mathcal{S}(F) :: \Psi & \longrightarrow & \mathcal{S}(F) \triangleright G' :: \Theta(\gamma) \end{array}$$

Fig. 1. The secure outsourcing diagram

Therefore, to make the outsourcing in Fig. 1 work it suffices to design a successful rework $\mathcal{R}$ such that $F \triangleright \mathcal{R}(G') :: \gamma$ holds for an arbitrary program component G' developed by external developers, provided that G' satisfies $\mathcal{S}(F) \triangleright G' :: \Theta(\gamma)$. Stated more formally, the definition of a secure mock is as follows.

Definition 4 (Secure mock)**.** Let F be a program component, Ψ be a security policy and γ be a specification. A *secure mock* of F is a test double $\mathcal{S}(F)$ such that the next proposition holds.

$$\forall G'. (\mathcal{S}(F) \triangleright G' :: \Theta(\gamma) \Rightarrow F \triangleright \mathcal{R}(G') :: \gamma) \wedge \triangleright\mathcal{S}(F) :: \Psi \quad (\clubsuit)$$

where, $\Theta(\gamma)$ denotes a modified specification of γ and $\mathcal{R}$ denotes a mapping between programs.

From the above definition, the secure mock construction problem is defined as the following constraint satisfaction problem.

Definition 5 (Secure Mock Construction)**.** Let F be a program, γ be a specification, and Ψ be a specification of security policy. The *secure mock construction problem* is configured with a triplet (F, γ, Ψ) and defined as the problem of finding $\mathcal{S}(F) \in \mathsf{Prog}, \Theta(\gamma) \in \mathsf{Spec}$ and $\mathcal{R} \in \mathsf{Prog} \rightarrow \mathsf{Prog}$ such that Eq. (♣) is satisfied.

Example 2 (Secure mock with dummy parameter)**.** Let F be a program component defined in Table I. Let γ be the predicate:

$\forall n \in \text{Name}, a \in \text{Age}, r \in \text{Residence}.\mathbb{R}(n, a, r, F(a, r))$ where $\mathbb{R}$ denotes the predicate representing R0-R3 from Section II. Let the security policy be $\Psi(F, \text{Residence})$ from Definition 3.

Here, we introduce a dummy parameter $d \in D = \{\alpha, \beta\}$. For a secure mock construction problem (F, γ, Ψ), we may solve the constraint by synthesizing a secure mock $\mathcal{S}(F)$ shown in Table II, and the following $\Theta(\gamma), \mathcal{R}$.

$$\Theta(\gamma) = \forall n, a, d \in \text{D}, r.\mathbb{R}(n, a, r, \mathcal{S}(F)(a, r, d))$$

$$\mathcal{R}(G')(n, a, r) = \begin{cases} G'(n, a, r, \beta) & \text{if } r = \text{Buzz City} \wedge a > 50 \\ G'(n, a, r, \alpha) & \text{otherwise} \end{cases}$$

TABLE II
A SECURE MOCK OF TABLE I

a:**Age**	**Residence**	**D**	**Premium**
$a \geq 0$	Foo Town	α	1,000
$a \geq 0$	Foo Town	β	1,500
$0 \leq a \leq 50$	Buzz City	α	1,000
$0 \leq a \leq 50$	Buzz City	β	1,500
$a > 50$	Buzz City	α	1,000
$a > \mathbf{50}$	**Buzz City**	β	**1,500**
$a \geq 0$	Barpolis	α	1,000
$a \geq 0$	Barpolis	β	1,500

$\Psi(\mathcal{S}(F), \text{Residence})$ is satisfied because the dummy parameter D hides the effect of the Residence parameter. For an arbitrary program component G' such that $\mathcal{S}(F) \triangleright G' :: \Theta(\gamma)$, we have $F \triangleright \mathcal{R}(G') :: \gamma$ because of the following reason. The conditional branch of $\mathcal{R}$ sets d to β for $r = \text{Buzz City} \wedge a > 50$. This allows the product owner to pick the part of the deliverable component G' corresponding to the row highlighted in bold in Table II. Otherwise, the branch sets d to α, allowing the product owner to select the part corresponding to the unhighlighted non-grayed-out rows. Therefore, $\mathcal{R}$ restores implementation for input domain restricted by the domain of Table I and the secure mock satisfies γ under the component F.

Let us use the above example to illustrate the difference in the three schools of TDD: the prior Detroit and London schools, and the new Hachioji school that we propose. In the Detroit school, since tests are conducted ideally with the actual program component, the test double of F, $\mathcal{S}(F)$, is simply F itself. It does not satisfy Ψ. Therefore, the proprietary knowledge may be leaked to the external developers and the security may be compromised. In the London school, tests are conducted with a test double of F that is not guaranteed to satisfy Eq. (♣). For instance, it may use a test double that just outputs a constant dummy value such as $\mathcal{S}(F)(a, r) = 1000$. This is secure (in the sense that it satisfies Ψ), but it makes it difficult for the developers to verify the requirements R0-R3 and may hamper their productivity. By contrast, the Hachioji school that we propose allows the use of a secure mock $\mathcal{S}(F)$ like the one shown in Example 2 that satisfies Eq. (♣) and guarantees both security and productivity. We conclude the section with some open problems.

Problem 1 (Determining Security Policy)**.** *Determining an appropriate security policy and formalizing as a predicate Ψ is an open problem.*

Problem 2 (Secure Mock Construction)**.** *Designing an algorithm for solving the constraint satisfaction problem of Definition 5 is another open problem.*

IV. Related Work

Software security is a long-standing problem in computer science. Many ideas have been proposed, such as methods for information flow control (IFC) [28]–[30] to formally prevent information leakage, hardware-based solutions such as trusted execution environments (TEE) [31]–[34], software obfuscation techniques [1], [2], privacy ensuring methods such as differential privacy [35], [36], software engineering disciplines for developing secure software such as secure development lifecycle (SDL) [37], and secure multi-party computation (SMPC) [38], to name a few. However, these prior works have primarily focused on the security of the completed software. By contrast, we aim to address the challenge of *information leakage during the software development process*.

Direct access to the source code of a proprietary component can be prevented by obfuscation or an approach like MORDEn [16]. However, as we have shown, that alone can be insufficient for securing semantic proprietary knowledge. Our work borrows fundamental concepts from the IFC research, where the problem is to prevent leakage of confidential high-security data to low-security outputs observable by unauthenticated users [25], [39], [40]. Goguen and Meseguer defined noninterference as the property that high-security data have no effect on low-security observations [25]. Quantitative information flow (QIF) is a framework for measuring the amount of information leakage and assessing the boundedness of the leakage amount [24], [30]. Our work adopted noninterference to formalize the strongest security policy of a secure mock, that is, we asserted that secret semantic characteristics of a program component, such as proprietary risk assessment knowledge, do not leak at all to external developers. We foresee that the QIF approach of measuring information leakage may play a key role in the development of a secure mock when a weaker security policy is sufficient.

V. Conclusion and Future Work

This paper introduced the Hachioji school as a novel style for secure outsourced software development and testing. Unlike the traditional London and Detroit schools, the Hachioji school centers on the use of secure mock, which is a mock component specifically designed to prevent confidential information leakage. We formalized the secure mock construction problem, clarifying how secure mocks can enable safe collaboration with external developers without exposing sensitive intellectual property.

Future work for the Hachioji school includes developing practical methods for constructing secure mocks under diverse security policies. Further research is needed to automate secure mock generation and to integrate the Hachioji school approach into real-world development pipelines. We hope that the Hachioji school and the secure mock concept will inspire new research at the intersection of software engineering, security, and formal methods, and provide a foundation for secure collaborative software development.

REFERENCES

[1] V. Balachandran and S. Emmanuel, "Software protection with obfuscation and encryption," in *Information Security Practice and Experience* (R. H. Deng and T. Feng, eds.), (Berlin, Heidelberg), pp. 309–320, Springer Berlin Heidelberg, 2013.

[2] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st ed., 2009.

[3] B. Boehm, "A view of 20th and 21st century software engineering," in *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, (New York, NY, USA), p. 12–29, Association for Computing Machinery, 2006.

[4] A. Begel, J. Bosch, and M.-A. Storey, "Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder," *IEEE Software*, vol. 30, no. 1, pp. 52–66, 2013.

[5] D. A. Tamburri, P. Lago, and H. v. Vliet, "Organizational social structures for software engineering," *ACM Comput. Surv.*, vol. 46, July 2013.

[6] "Github codespaces." https://github.com/features/codespaces. GitHub, Inc., Accessed: 2025-07-12.

[7] E. Estellés-Arolas and F. González-Ladrón-De-Guevara, "Towards an integrated crowdsourcing definition," *J. Inf. Sci.*, vol. 38, p. 189–200, Apr. 2012.

[8] S. Greengard, "Following the crowd," *Commun. ACM*, vol. 54, p. 20–22, Feb. 2011.

[9] K.-J. Stol and B. Fitzgerald, "Two's company, three's a crowd: a case study of crowdsourcing software development," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), p. 187–198, Association for Computing Machinery, 2014.

[10] M. N. Harrell, "Synergistic security: A work system case study of the target breach," *Journal of Cybersecurity Education, Research and Practice*, vol. 2017, Dec. 2017.

[11] D. McDaniel, "Toyota suffered a data breach by accidentally exposing a secret key publicly on github." https://blog.gitguardian.com/toyota-accidently-exposed-a-secret-key-publicly-on-github-for-five-years/, 2022. Accessed: 2025-07-12.

[12] C. Farinella, A. Ahmed, and C. Watterson, "Git leaks: Boosting detection effectiveness through endpoint visibility," in *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 701–709, 2021.

[13] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, "Detecting and mitigating secret-key leaks in source code repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, p. 396–400, IEEE Press, 2015.

[14] "Cycode." https://cycode.com/source-code-leakage-detection/. Cycode Ltd., Accessed: 2025-07-12.

[15] Flare, "Preventing & identifying source code leaks: A flare guide." https://flare.io/learn/resources/blog/preventing-identifying-source-code-leaks-a-flare-guide/, 2022. Flare Systems, Inc., Accessed: 2025-07-12.

[16] S. Saito, "Coding and debugging by separating secret code toward secure remote development," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, p. 1864–1869, IEEE Press, 2024.

[17] N. T. Courtois, G. V. Bard, and D. Wagner, "Algebraic and slide attacks on keeloq," in *Fast Software Encryption* (K. Nyberg, ed.), (Berlin, Heidelberg), pp. 97–115, Springer Berlin Heidelberg, 2008.

[18] F. D. Garcia, G. Koning Gans, R. Muijrers, P. Rossum, R. Verdult, R. W. Schreur, and B. Jacobs, "Dismantling mifare classic," in *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, (Berlin, Heidelberg), p. 97–114, Springer-Verlag, 2008.

[19] G. Koning Gans, J.-H. Hoepman, and F. D. Garcia, "A practical attack on the mifare classic," in *Proceedings of the 8th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications*, CARDIS '08, (Berlin, Heidelberg), p. 267–282, Springer-Verlag, 2008.

[20] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, (USA), p. 601–618, USENIX Association, 2016.

[21] D. Oliynyk, R. Mayer, and A. Rauber, "I know what you trained last summer: A survey on stealing machine learning models and defences," *ACM Comput. Surv.*, vol. 55, July 2023.

[22] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1st ed., 2009.

[23] K. Beck, *Test driven development*. The Addison-Wesley signature series, Boston, MA: Addison-Wesley Educational, Nov. 2002.

[24] M. Alvim, K. Chatzikokolakis, A. McIver, C. Morgan, C. Palamidessi, and G. Smith, *The Science of Quantitative Information Flow*. Information Security and Cryptography, United States: Springer, Springer Nature, 2020.

[25] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, pp. 11–11, 1982.

[26] E. Carmel and P. Tjia, *Offshoring Information Technology: Sourcing and Outsourcing to a Global Workforce*. Cambridge University Press, 2005.

[27] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *J. Comput. Secur.*, vol. 17, p. 517–548, Oct. 2009.

[28] E. Cohen, "Information transmission in computational systems," *SIGOPS Oper. Syst. Rev.*, vol. 11, p. 133–139, Nov. 1977.

[29] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, p. 504–513, July 1977.

[30] M. Bhargava and C. Palamidessi, "Probabilistic anonymity," in *CONCUR 2005 – Concurrency Theory* (M. Abadi and L. de Alfaro, eds.), (Berlin, Heidelberg), pp. 171–185, Springer Berlin Heidelberg, 2005.

[31] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, 2010.

[32] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-x: A flexible architecture for hardware-managed isolated execution," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 190–202, 2014.

[33] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: minimal hardware extensions for strong software isolation," in *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, (USA), p. 857–874, USENIX Association, 2016.

[34] M. Sommerhalder, *Trusted Execution Environment*, pp. 95–101. Cham: Springer Nature Switzerland, 2023.

[35] C. Dwork, "Differential privacy," in *Automata, Languages and Programming* (M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, eds.), (Berlin, Heidelberg), pp. 1–12, Springer Berlin Heidelberg, 2006.

[36] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Theory of Cryptography* (S. Halevi and T. Rabin, eds.), (Berlin, Heidelberg), pp. 265–284, Springer Berlin Heidelberg, 2006.

[37] S. Lipner, "Security development lifecycle," vol. 34, no. 3, pp. 135–137.

[38] D. Evans, V. Kolesnikov, and M. Rosulek, "A pragmatic introduction to secure multi-party computation," *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2–3, p. 70–246, 2018.

[39] J. McLean, "Security models and information flow," in *Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 180–187, 1990.

[40] D. McCullough, "Noninterference and the composability of security properties," in *Proceedings. 1988 IEEE Symposium on Security and Privacy*, pp. 177–186, 1988.