information

MDPI

Article

P2ISE: Preserving Project Integrity in CI/CD Based on Secure Elements †

Antonio Muñoz [1,*], Aristeidis Farao [2], Jordy Ryan Casas Correia [1] and Christos Xenakis [2]

[1] Computer Science Department, University of Malaga Campus de Teatinos s/n, 29071 Malaga, Spain; ryan@uma.es
[2] Department of Digital Systems, University of Piraeus, 18534 Piraeus, Greece; arisfarao@unipi.gr (A.F.); xenakis@unipi.gr (C.X.)
* Correspondence: amunoz@lcc.uma.es

† This Paper is an Extended Version of Our Paper Published in the Proceedings of the 1st Workshop on Dependability and Safety Emerging Cloud & Fog Systems (DeSECSys 2020), Entitled "ICITPM: Integrity Validation of Software in Iterative Continuous Integration through the Use of Trusted Platform Module (TPM)".

Abstract: During the past decade, software development has evolved from a rigid, linear process to a highly automated and flexible one, thanks to the emergence of continuous integration and delivery environments. Nowadays, more and more development teams rely on such environments to build their complex projects, as the advantages they offer are numerous. On the security side however, most environments seem to focus on the authentication part, neglecting other critical aspects such as the integrity of the source code and the compiled binaries. To ensure the soundness of a software project, its source code must be secured from malicious modifications. Yet, no method can accurately verify that the integrity of the project's source code has not been breached. This paper presents P2ISE, a novel integrity preserving tool that provides strong security assertions for developers against attackers. At the heart of P2ISE lies the TPM trusted computing technology which is leveraged to ensure integrity preservation. We have implemented the P2ISE and quantitatively assessed its performance and efficiency.

Keywords: CI/CD pipeline; code integrity; trusted computing; TPM

check for updates

Citation: Muñoz, A.; Farao, A.; Correia, J.R.C.; Xenakis, C. P2ISE: Preserving Project Integrity in CI/CD Based on Secure Elements. *Information* **2021**, *12*, 357. https://doi.org/10.3390/info12090357

Academic Editor: Willy Susilo

Received: 30 June 2021
Accepted: 23 August 2021
Published: 31 August 2021

1. Introduction

Lately, the programming community has been witnessing a rapid increase in the adoption of development methods like Development and Operations (*DevOps*), *Agile* and Continuous Integration, and Continuous Delivery (*CI/CD*) by developers across the world. Automation is a key aspect of the aforementioned methods, used to build, deliver, and test high-frequent increments of features [1–3]. *DevOps* has been defined as a set of practices intended to optimize the time required between committing a change to the system and for said change to be incorporated into production code. *Agile* practices are focusing on eliminating the aforementioned processes and accelerating product delivery, by quickening the development life cycle. The *CI/CD* pipeline is considered to be among the best practices for delivering code changes more frequently and reliably during code implementation. On one hand, Continuous Integration (*CI*) can be described as the guided practices that enable continuous surveillance in code repositories allowing development teams to implement changes in code and their check-in. To achieve this, relevant mechanisms are required for the integration and validation of code changes derived from multi-platform features from contemporary applications. Technically, we can define *CI*'s primary goal as providing a set of tools to build, package and test applications in an automated and consistent way. This consistency allows the teams to increase the frequency of committing code changes, improving both collaboration and software quality. On the other hand, the

Continuous Delivery (*CD*) technique which picks up at the end of *CI*, performs automation in application delivery to particular infrastructures. The use of different environments for code production, development, and testing where multiple code changes are submitted at the same time has recently become a widely popular practice. *CD* provides an automated way to perform those changes, keeping stored packaging parameters bound to every delivery.

Due to its features, *CI/CD* is one of the most popular practices used by software developers to deliver code changes in the most reliable way. According to a survey by DigitalOcean [4] on developer trends released in 2017, it has been revealed that 42% of the survey respondents and members of the DigitalOcean developer community use a *CI/CD* solution and they believe that its most beneficial aspect is that it allows developers to quickly review and deploy code. The *CI/CD* pipeline consists of specific components; however, it inherits the security considerations which are related to the traditional IT system but also connected to human behavior. Establishing mechanisms that protect the integrity of a software project against cyber-attacks that threaten to compromise it is of paramount importance for ensuring the robustness of the final product. Despite the increasing popularity of *CI/CD* tools among the developers community and the all attention they have been getting, to the best of our knowledge, there is no work in the literature proposing a way to guarantee the integrity of software projects as part of the *CI/CD* pipelines. This paper identifies and analyzes the security gap that exists in the *CI/CD* pipeline regarding a software project's integrity. To this end, we propose the *P2ISE*, a novel tool that is tailored to the *CI/CD* concept and employs trusted computing technologies, such as secure elements, to ensure the integrity of software projects. More specifically, at the heart of *P2ISE* lies the *TPM* trusted computing technology that enables secure storage of critical data (e.g., cryptographic keys), as well as secure execution of sensitive operations. The proposed *P2ISE* has been designed taking into account the existing, traditional architecture of the *CI/CD* pipelines, which extends by introducing a new entity responsible for ensuring the integrity of each software project. To assess the performance of *P2ISE*, we have fully implemented and deployed a prototype utilizing a real *TPM* which was used to evaluate the median duration time, CPU utilization, and memory consumption of the *P2ISE* processes against various software projects. Numerical results show that *P2ISE* can efficiently operate without depleting developers' resources. In summary, the paper makes the following contributions:

- Define security and functional requirements of a tool that is meant to provide developers with *CI/CD* features following a security by design approach.
- Propose *P2ISE*, a solution for integrity preservation for software projects within *CI/CD* environments based on the use of secure elements, in particular the *TPM* chipset. To the best of our knowledge, this is the first paper that proposes a tool to bridge the identified security gap.
- Assess the proposed *P2ISE*'s performance and qualitatively reason about its security properties. For this purpose, we have implemented and evaluated it against various projects.

The paper unfolds as follows: Section 2 presents essential background information on *CI/CD*, the motivation of our work, the threat analysis we have performed on the *CI/CD* pipeline, and finally the security and functional requirements we have identified based on the developers' needs. Next, Section 3 discusses the related work, whereas Section 4 elaborates on the processes of the *P2ISE* describing in detail all the required steps. Section 5 includes a quantitative performance evaluation of *P2ISE*, and Section 6 discusses its security properties. Lastly, Section 7 concludes the paper.

2. The CI/CD Concept

2.1. Definition and Participants

The *CI/CD* objective is to enable developers to deliver code changes as frequently as needed, in the most reliable manner. For this reason, *CI/CD* foresees continuous testing, which typically is offered as performance, regression, and another set of tests done within a *CI/CD* pipeline. Developers submit their code for commitment into the version control repository. Also, it is common practice to establish a minimal rate of daily code commitments per team to facilitate the identification of defects and bugs on smaller delta pieces of code rather than large-scale developments. Moreover, working on smaller commit cycles reduces parallel working on the same code by multiple developer teams. Many teams that implement continuous integration start with version control configuration and practice definitions. Even though checking in code is frequently performed, features and fixes are implemented in both short and longer time frames.

Different techniques are used to control and filter code for production in *CI*. Among the most common practices requires from the developers to run regression tests in their environments, which implies that only code that passed regression tests was committed. We notice that commonplace for development teams is to have at least one development and testing environment, which allows for reviewing and testing application changes. A *CI/CD* tool such as Jenkins (https://jenkins.io/ (accessed on 22 August 2021), CircleCI (https://circleci.com/ (accessed on 22 August 2021), AWS CodeBuild (https://aws.amazon.com/es/codebuild/ (accessed on 22 August 2021), Azure DevOps (https://docs.microsoft.com/en-us/azure/devops/?view=azure-devops (accessed on 22 August 2021), Atlassian Bamboo (https://www.atlassian.com/software/bamboo (accessed on 22 August 2021), or Travis CI (https://travis-ci.com/ (accessed on 22 August 2021) is used to automate the steps and provide reporting. A typical *CD* pipeline [2] includes the following stages: (i) built; (ii) test; and (iii) deploy. Nonetheless, improved pipelines include also the following stages: (i) picking code from version control and executing a build; (ii) allowing any automated action such as restarting or shutting down both cloud infrastructure, services, or service endpoints; (iii) moving code to the target computing environment; (iv) setting up and managing environment variables; (v) enabling services as API services, database services or web servers to be pushed to application components; (vi) allowing rollback environments and the execution of continuous tests and (vii) alerting on delivery state and data log are provided. A *CI/CD* environment consists of (i) the *Source Code Control Server* which is responsible to manage changes to the project's documents (ii) the *Assembly Server* which receives the changes and assembles them; (iii) the *Testing Server and Deployment Server* that validates the project work and then publishes the latest version. Conceptually each previously mentioned server is located on different premises.

2.2. Motivation

Software development has radically evolved in the last years, from classical rigid models like the waterfall to Agile methodologies providing less docking among member functions developments and more oriented towards impending automation demanded by Industry 4.0 [5]. However, the related security requirements elicited from the procedures followed in recent models have not been carefully addressed. DigitalOcean [6] published as part of a *CI/CD* best practices tutorial that the proper way to ensure a *CI/CD* environment for a company devoted to virtual server deployment under premises is the isolation from external access. Protecting the *CI/CD* server is crucial, and for that purpose several solutions exist, such as the use of secure shell (SSH) or private keys for APIs connecting through services like GitHub (https://github.com/ (accessed on 22 August 2021) or GitLab (https://about.gitlab.com/ (accessed on 22 August 2021) to the *CI/CD* environment. Moreover, the use of a strong password and a 2-factor authentication solution is also widely recommended [7]. However, Milka [8] revealed that less than 10% of Google users make use of a 2-factor authentication solution. A fail in securing those keys could lead to source code filtering or code modifications as a result of impersonation attacks.

Furthermore, *CI/CD* solutions provide an intermediate interface to manage *Assembly and Test Server* (i.e., Jenkins or GitLab) through a web interface. In the case of Jenkins, it is enabled as credential-based access, and thus, the security of this interface is another issue to consider. We notice that many providers ignore recommendations about *CI/CD* server isolation. Also, Paul et. al in [9] revealed that developers who work with *CD* pipeline are only familiar with the general security attributes and lack in-depth security knowledge. As described in [6], failures in a *CI/CD* pipeline are immediately visible and could halt the advancement of the affected release to the later stages of the cycle.

Nowadays, dockerization and virtualization are used to protect against unexpected events. However, currently deployed software is not considered trustworthy because, on most occasions, software security measures are not carefully considered. Deployment tends to be isolated in host machines, restricting privileges and hardware access as much as possible; however, it is controversial whether developers can rely on these measures or not. The underlying software that controls these virtual machines acting as an intermediary layer between every virtual machine and the hardware is the *Hypervisor*. Dedicated to handling virtual machines, *Hypervisor* can become a single-point-of-failure. For instance, an attacker who gained control of *Hypervisor* can handle every virtual machine without leaving any trace that could reveal the source of the attack. This technique is known as *hyperjacking* [10], and its most common implementation is to insert a malicious *Hypervisor* to replace the original one. The above is an example of a deployment pipeline attack scenario; however, there are many possible attack scenarios. Figure 1 depicts how this attack could be implemented in four steps: (i) *Developer* implements a new feature and this is uploaded to *Source Code Control Server* (*Git*-based server in most cases); (ii) changes finished in *Source Code Control Server* are sent to *Assembly and Test Server;* (iii) *Assembly and Test Server* assembles a new software version and conducts unitary test and linkage prepared for this software and (iv) once recommended tests are passed, a new version of the software is made public (deployment).

However, assuming that every communication between the participants is secure, we have identified that the most vulnerable participant is the *Assembly and Test Server*. In most cases, it is considered trusted because its interaction is restricted to insert source code. Notwithstanding, we have identified a security gap in a process that is described below. For example, we assume the existence of a malicious agent that has been granted access to *Assembly and Test Server* and inserts a piece of code for detecting every time source code is generated and files are modified. Then it replaces a piece of a key source code file opening a backdoor. The changes will be deployed since at this point the source code is considered as checked and valid. Once the deployment is done, then attacker can complete his attack.

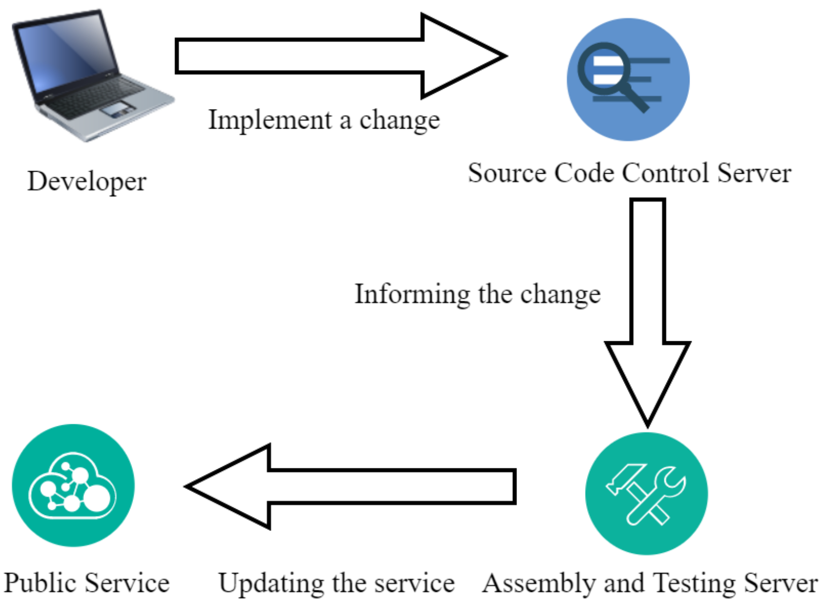

Figure 1. Identified Risk in Continuous Integration Process.

2.3. Threat Analysis

To identify all possible threats for a *CI/CD* pipeline (see Figure 1) we utilize Microsoft's threat modeling tool that supports the STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege) methodology. The scenario that we draw contains the basic entities that participate in a *CI/CD* pipeline, which are the following: (i) *Developer*; (ii) *Source Code Control Service*; (iii) *Assembly and Test Server* and (iv) *Public Service*. We assumed that the *Developer* is a human and does not authenticate himself. Moreover, the *Source Code Control Service* uses both authentication and authorization mechanisms. While the *Assembly and Test Server* utilizes only an authorization mechanism. Finally, the *Public Service* is represented as a Web Service and presents the relevant updates. Defining this architecture, we can observe that the assets in this scenario are distinguished in Information and Physical assets; these are summarized in Table 1. However, at this point we have to mention that the identified threats (see below) are related with the specific architecture (see Figure 2). By this, we mean that a different use case may generate different threats applicable to our scenario.

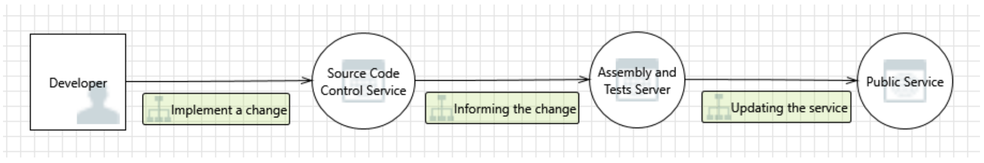

Figure 2. CI/CD process scenario in Microsoft Threat Modeling Tool.

Table 1. CI/CD assets.

Information Assets	Physical Assets
User credential	Server
Authorization mechanism	Computer (Developer's PC)
Log information	
Project code	
Product (*Public Service*)	

By applying the STRIDE methodology to the aforementioned scenario, we identified the threats that are presented below.

T1. *Elevation Using Impersonation*: *Source Code Control Service* may be able to impersonate the context of a Developer in order to gain additional privilege.

T2. *Elevation Using Impersonation*: *Assembly and Test Server* may be able to impersonate the context of *Source Code Control Service* in order to gain additional privilege.

T3. *Weak Authentication Scheme*: Custom authentication schemes are susceptible to common weaknesses such as weak credential change management, credential equivalence, easily guessable credentials, null credentials, downgrade authentication or a weak credential change management system.

T4. *Source Code Control Service Process Memory Tampered*: If *Source Code Control Service* is given access to memory, such as shared memory or pointers, or is given the ability to control what *Assembly and Test Server* executes (for example, passing back a function pointer.), then *Source Code Control Service* can tamper with *Assembly and Test Server*. Consider if the function could work with less access to memory, such as passing data rather than pointers. Copying data provided, and then validate it.

T5. *Collision Attacks*: Attackers who can send a series of packets or messages may be able to overlap data. For example, packet 1 may be 100 bytes starting at offset 0. Packet 2 may be 100 bytes starting at offset 25. Packet 2 will overwrite 75 bytes of packet 1.

T6. *Assembly and Test Server Process Memory Tampered*: If *Assembly and Test Server* is given access to memory, such as shared memory or pointers, or is given the ability to control what *Public Service* executes (for example, passing back a function pointer.), then *Assembly and Test Server* can tamper with *Public Service*. Consider if the function could work with less access to memory, such as passing data rather than pointers. Copy in data provided, and then validate it.

T7. *Replay Attacks*: Packets or messages without sequence numbers or timestamps can be captured and replayed in a wide variety of ways. Implement or utilize an existing communication protocol that supports anti-replay techniques (investigate sequence numbers before timers) and strong integrity.

T8. *Elevation Using Impersonation*: *Public Service* may be able to impersonate the context of *Assembly and Test Server* in order to gain additional privilege.

T9. *Cross Site Scripting*: The web server *Public Service* could be a subject to a cross-site scripting attack because it does not sanitize untrusted input.

2.4. Security and Functional Requirements

As previously mentioned, the *CI/CD* is a concept that provides many advantages to developers but also attracts the attackers' attention. Moreover, a *CI/CD* ecosystem inherits the risks (see Figure 1) of traditional IT systems as well as the risks posed by the developers' poor security practices. This leads to the conclusion that security and functional requirements need to be clarified. Since the functional requirements of *CI/CD* ecosystems have been well-established by the literature, lately the focus appears to be shifting towards the security-related conditions that must be met by every proposed solution. For solutions designed to address the needs of developers who employ the *CI/CD* pipeline, we define the following security and functional requirements after considering the *CI/CD* components, users' security and functional demands, and the related research. We have to notice that we redefine functional requirements following security by design approach.

2.4.1. Security Requirements

Since a *CI/CD* ecosystem involves risks inherited from both traditional IT system and developers' poor security practices, we re-establish the set of standard security requirements applied to it [11–15]. In addition, in "Who is Using Jenkins" ((https://wiki.jenkins.io/pages/viewpage.action?pageId=58001258 (accessed on 22 August 2021) there are projects as KDE (https://kde.org/ (accessed on 22 August 2021), Apache (https://www.apache.org/ (accessed on 22 August 2021), AngularJS (https://angularjs.org/ (accessed on 22 August 2021) and Ubuntu (https://ubuntu.com/ (accessed on 22 August 2021) that are publicly accessible and may offer significant help in the integration of general security requirements.

S1. *Data confidentiality*: Code of a project within the *CI/CD* environment should be available only to responsible developers. No adversaries should be able to read and edit the code of the software project.

S2. *Data integrity*: All code transactions (e.g., *push* commands) among the engaging entities (developers) should be protected against malicious alternations. Each process should be monitored and verified.

S3. *Non-repudiation*: Once a developer completes an action (e.g., code changes) then he should not by able to deny it; each actor should be responsible for his actions. This lead to the fact that each action should be monitored and securely recorded.

S4. *Accountability*: A developer should be held accountable for his actions.

2.4.2. Functional Requirements

Apart from the security requirements, a *CI/CD* pipeline has processes that require specific functionalities to be enabled. Analyzing the literature literature [12,15], we redefine the established requirements following the developing norms and a security by design approach.

F1. *Passive storage with no shared access*: Data that should be accessed only by entities that have been authorized by the owner for specific actions needs to be protected against access attempts by unauthorized entities or to unauthorized actions, while maintaining availability for authorized users.

F2. *Privileged activity tracking*: All modification attempts should be monitored.
F3. *Integrity verification*: Each modification attempt should be verified via hash function before being deployed for avoiding malicious activities.
F4. *Key utilization*: The keys that are used for modifications should have been created and stored only inside the *TPM* preventing possible hardware attacks and data leakage.
F5. *Time consuming*: Since the deployment of project modifications depends on the project's size, the time added due to the extra verification should not negatively affect the *CI/CD* performance.

3. Related Work

The literature in the field of P2ISE contains the security approaches in CI/CD environments and the technology of the secure element.

3.1. Security Approach in CI/CD Environment

While there is a plethora of works that highlight the importance of security in *CI/CD* pipelines, to the best our knowledge, this is the first paper that proposes the incorporation of an integrity-preserving method in the *CI/CD* pipeline, leveraging for this purpose trusted computing technologies. This work is an extended version of the paper entitled "ICITPM: Integrity Validation of Software in Iterative Continuous Integration Through the Use of Trusted Platform Module (TPM)" by Muñoz et al., that has been published in the proceedings of the 1st Workshop on Dependability and Safety Emerging Cloud & Fog Systems (DeSECSys 2020) [16]. In particular, this extended version includes: (i) a detailed model of possible threats in the study architecture; a precise justification of the need to use a secure element as a trust anchor; (ii) a summary of different benchmarking tests that have been carried out to analyze different projects that exhibit a variety in terms of size and conditions, which demonstrate with real figures the impact that the proposed solution has in terms of performance; and (iii) a discussion related to the security features of *P2ISE*. Parts of the work in [16] are reused in the present paper.

As mentioned above, the use of *CI/CD* has become a prominent practice within the software development community. There are different works such as [17] that review some of the most commonly used practices for *CI/CD* with a specific provider (Azure Kubernetes), while others focus on the use of proprietary tools such as GitOps for a Kubernetes *CI/CD* pipeline. Other works propose ways for organizations to incorporate security practices in the CD process [18] and the separation of duties with the consequent division of development, security and operations roles (*DevSecOps*), by introducing automation mechanisms that reduce the need for a human interface, or by using a development framework for Trusted Execution Environments (TEE) on top of deployment artifacts for their protection [11]. Moreover, authors in [6] proposed a gatekeeping mechanism that safeguards the most important environments from untrusted code through a physical separation between the *Testing Server* and *Assembly Server*. This, nevertheless, produces a false sense of security since the integrity of the source code is not guaranteed.

We have to note that *P2ISE* is not the first work that utilizes secure elements to reinforce the security of the *CI/CD* pipeline. The integration of secure elements has also been suggested in previous works. Despite the effort for different TEE implementations, such as ARM TrustZone, Intel SGX and recently AMD SEV, to be introduced and leveraged in the software development process, TEEs have so far been more prominent on mobile devices [19]. The proposal from Asylo [11] achieves a breakthrough in improving the *CI/CD* process by integrating an additional step so that artifacts are protected against untrusted administrators, achieving high level of protection even from cloud service providers. Yet, it does not provide a solution to the gap identified and solved in this work. Also, Bass et al. [13] proposes an engineering process within trusted components embedded in parts of the pipeline, which is intimately related to our approach although the use of trusted hardware is not foreseen. Moreover, in [14] different security tactics have been applied between *CD* components communications with encouraging results, whereas

Rimba et al. [15] have presented an approach based on the use of composing patterns to address security issues in *CD* pipeline.

Moreover, there are approaches as Nomad [20], Mood et al. present a defense system against known and future side channels and Deepa et al. [21] deal with securing web applications from injection and logic vulnerabilities or approaches based on static analysis and run-time protection and mitigation of vulnerability impact based on security testing techniques [22]. Lipke [23] studies threats in *CD* pipeline using the *STRIDE* methodology implementing a proof of concept based on Docker. Schneider [24] proposes a four-staged dynamic security scanning methodology (pre-authentication scanning, post-authentication scanning, back-end scanning and scanning workflows specific to the targeted application). Also, the same author introduces the *SecDevOps* Maturity Model(SDOMM). This can be considered as instructions for automatically achieving particular security aspects in *CI* pipeline.

In summary, the related work on the security issues of the *CI/CD* pipeline copes with various emerged challenges. However, the preservation of integrity is a requirement that has not been met yet. Therefore, a new scheme dedicated to the integrity preservation is required. The proposed *P2ISE* aims to bridge the gap and enhance the security level of the *CI/CD* pipeline in general.

3.2. Secure Element as Trust Anchor

The technological pillar of the proposed solution that provides indisputable security properties is a secure element (SE) with the role of trust anchor. Our concept of secure element is a by design protected from unauthorized access microchip with features as data storing and secure running of applications inside itself. SE can typically be found as a dedicated chip installed on the motherboard of a device (i.e. a smartphone), in an external element such as a flash memory card, in the circuitry of devices such as the SIM card itself used in mobile phones, or as a cloud service in Host Card Emulation technology. A new family of embedded environments known as Trusted Execution Environments (TEE) [25,26] has emerged. A TEE is a hardware environment with a secure operating system that is isolated and completely separated from the mobile platform. The concept behind a TEE implementation is to provide an independent execution environment that runs alongside the operating system [27]. This environment provides certain security services to the native operation system [28]. Over the last few years, work has been ongoing to standardize the TEE architecture itself as well as the interfaces to interact with environments such as secure environments and SE led by GlobalPlatform (https://globalplatform.org/ (accessed on 22 August 2021). The main objective of this standardization is to provide a hardware and software environment for securing applications such as banking or corporate applications. Moreover, Matetic et al. [29] propose a flexible delegation system with a TEE-based implementation on any browser-based device (smartphone, laptop, desktop, tablet, etc.) that can be also considered SE. In the case of TEE, two implementations have been taken as reference, the proposal of Intel SGX and the proposal of TrustZone and the Global Platform TEE implementation. These two implementations have been taken as references, since the range of TEE capabilities is very wide and each alternative offers different sets of features, but these are widely used and representative of different approaches.

Since specific requirements have been extracted (see Section 2), we have decided to integrate the implementation of Infineon's TPM as a technology of the Trusted Computing standard. The Trusted Platform Module (*TPM*) is useful for data protection, as well as for the generation of platform integrity tests, which for our case is a basic feature. However, *TPM* devices are known to come with certain restrictions. Among them, the most significant one is the investment required for this device. For these reasons we have include a detailed comparison among this *TPM* and TEE implementations to support our decision.

Vasudevan et al. [30] describe the following TEE objectives: (i) Isolated Execution; (ii) Secure Storage; (iii) Integrity, Confidentiality; (iv) Freshness; (v) Remote Attestation; (vi) Secure Provisioning, and (vii) Path of Trust. Among these required properties for any TEE we have to assume that it is difficult to find all of them in the same TEE implementation. Another *TPM*'s advantage is that its specification is open with all that this entails in terms of transparency and evolution, while the Intel SGX implementation is a closed one.

Regarding the functional requirements described above (see Section 2), passive storage with no shared access can be achieved using both *TPM* and TEE. However, we have to consider the number of vulnerabilities found in TEE implementations as those focus on the isolation between worlds [31,32], the wide attack surface [33,34] and memory side-channel attacks [35–42]. Moreover, there are side-channel attacks focusing on the TEE's covert channel communication Prime+Probe [35], Evict+Time [35], Flush(Evict)+Reload [36] and Flush+Flush [37]. Therefore, we can safely deduce that *TPM* is a more robust alternative than the TEE.

Furthermore, the activity tracking requirement can be addressed using both *TPM* and TEE. However, the integrity verification related to that required tamper-proof resistant feature to avoid possible attacks as meltdown [43] and spectre [44] is provided only from *TPM*. As keys are issued and stored within the *TPM*, this feature contributes on building the integrity required. The last functional requirement to be considered is the time consumption. However, in terms of efficiency, *TPM* is less efficient since it only has a slow communication bus with the CPU. while in TEE the code is executed directly on a more powerful main CPU, giving a higher level of efficiency, being faster as well as having access to all the RAM available to the OS at the time of execution.

In terms of functionality, the design of the *TPM* states that the processor of the module itself remains isolated from the CPU. For this reason, the *TPM* can only operate with what is provided to it, i.e., it is a passive device that must be accompanied by certain software to make use of its functionality. Indeed, an additional software is needed but some implementations as tpm2 (https://github.com/tpm2-software (accessed on 22 August 2021) and xaptum (https://github.com/xaptum (accessed on 22 August 2021) follow the recommendations set out by Trusted Computing Group (https://trustedcomputinggroup.org/ (accessed on 22 August 2021). Among the different TEE alternatives, TrustZone only allows an isolated section. In the case of Intel SGX, there is a strong linkage to the CPU that allows it to control the management of virtual memory, context switches, as well as high-speed communications. On its part, *TPM* functionality is fully integrated into the hardware and although its design is aimed at providing flexibility. Also, flashing allows arbitrary code to be executed, but has no access to the operating system or drivers. Therefore, only computation and very simple I/O is possible.

Finally, protection against physical attacks is a mandatory requirement; however, TEE does not provide protection against physical attacks. SGX solution provides a protection mechanism against this category of attacks. However, certain weaknesses have appeared, such as the interface to the CPU which is not protected at all, the trusted zone keys may be in unencrypted flash memory or the SGX keys may be in the CPU, which should not be trivial to extract. In contrast, the *TPM* guarantees the physical protection of the keys, the model is much more robust and secure and in spite of the additional cost and other mentioned restrictions we consider the suitable choice to our proposed tool to bridge the integrity gap previously described (see Section 2).

4. The P2ISE Concept

In this section, we present a blueprint of the proposed tool's architecture along with the process that takes place for its seamless integration with a *CI/CD* platform as shown in Figure 3. In *P2ISE*, apart from the standard entities that participate in a *CI/CD* pipeline (see Section 2) we introduce the *Trusted Integrity Platform* (*TIP*) (see also Table 2), an additional entity that we consider the pillar of our scheme. *TIP* is a server equipped with a *TPM* and a trust software stack for testing the software project integrity. The *TPM* is used as the anchor for integrity and validation proofs as it provides guarantees for building a robust *TIP* server with a controlled software stack that *P2ISE* leverages so it can assure that no malicious code can alter the project. The integrity of the *TIP* server is secured by the *TPM* public key, since its trusted boot process is bound to the corresponding *TPM* sealed key.

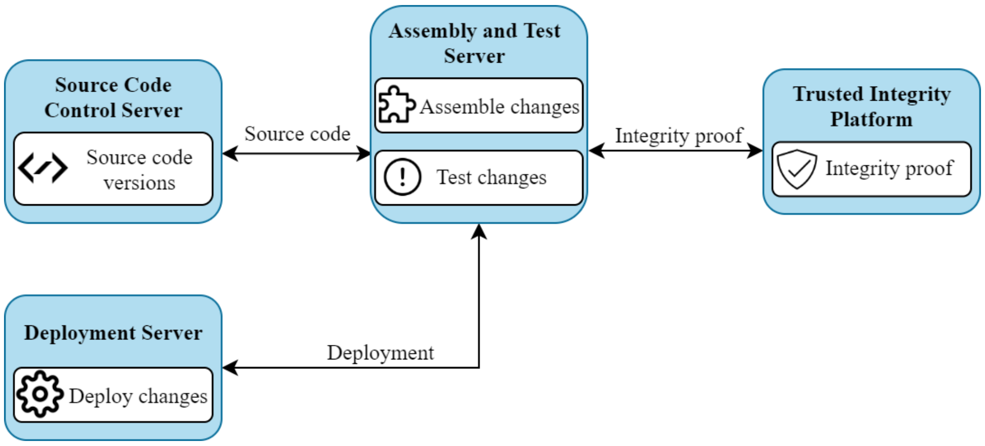

Figure 3. Architectural components.

Table 2. Main entities participating in *P2ISE*.

Entity	Description
Developer	A developer who initiates commands.
Source Code Control Server	Track changes in source code.
Assembly and Test Server	Receives changes and assembles them.
Deployment Server	Deploy changes.
Trusted Integrity Platform	Proves software project's integrity.

4.1. P2ISE

P2ISE consists of three individual integrity proofs. The first one is taken before installing all the required software dependencies and guarantees integrity between code instances from the *Assembly and Test Server* and *Source Code Control Server*. The second integrity check guarantees that source code under *Assembly and Test Server* remains unchanged from external agents. The third validation checks that the whole process was successfully completed and the code remains unchanged after the assembling.

Regarding the high-level design of the proposed solution, the underpinning idea is that the *TIP* will safeguard the integrity of a *CI/CD* pipeline establishing a secure and trustworthy code integrity control when an assembly code computer is not trusted, utilizing the *TPM* technology. One of the novelties of *P2ISE* lies in the fact that we propose a 3-factor security check. In particular, the third security check provides strong security assertions, since it utilizes the *TPM* keys that are safely stored in the module. *P2ISE* provides a set of functionalities related to software integrity where trust is by default ensured thanks to the use of the *TPM* trusted technology. Moreover, *P2ISE* follows an user-centric approach. Developers are responsible for submitting their code for commitment to a corresponding *Source Code Control Server* and assumed to be trustworthy by the owner of a specific software project. However, *Developers* have always been susceptible to different kind of attacks or bad security practices, making them the weakest link in a *CI/CD* ecosystem.

We consider that a 3-factor integrity proof [45,46] is the most appropriate for the *CI/CD* pipeline. A comprehensive description of the complete process is described below:

- **First integrity proof measure**: is taken before installing all dependencies required for the project; this guarantees that the source code from the *Assembly and Test Server* is identical to the *Source Code Control Server*.
- **Second integrity proof measure**: it guarantees that source code under assembly remains unchanged from external agents in *Assembly and Test Servers*.

- **Third integrity proof measure**: it guarantees that the whole process was successfully completed without undesired modifications after the project was assembled.

Figure 4 shows a sequence diagram with *TIP* process communications in the *CI/CD* pipeline. This shows the 3-factor verification described above, as well as the check point of every integrity proof. The algorithm that enables communication with the *TIP* server actually implements project integrity validation. This script is based on PowerShell and it is tested on Jenkins. The procedure script is included as part of *CI/CD* pipeline testing batches. Also, we have included the *TIP* server script communication from Jenkins in PowerShell.

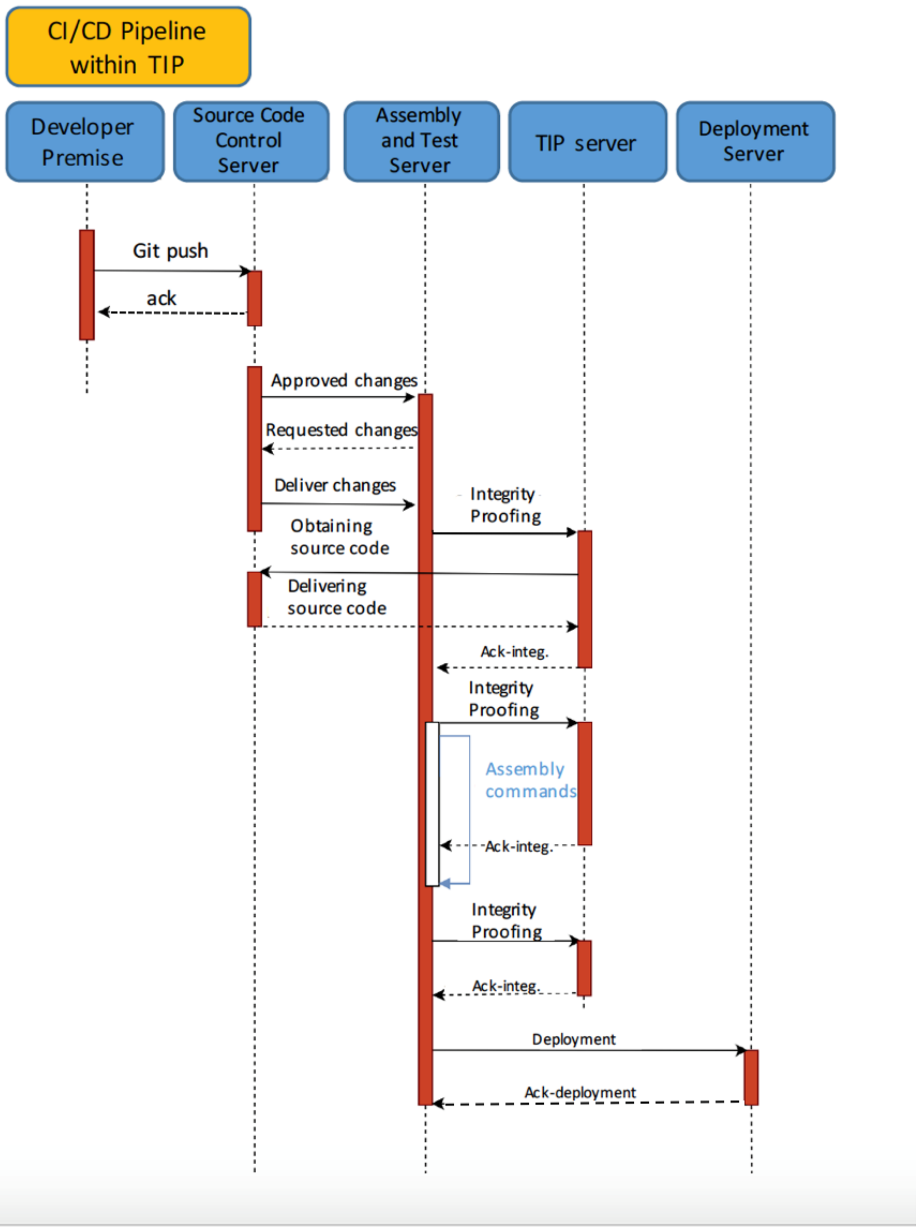

Figure 4. Sequence diagram *CI/CD* pipeline within the TIP server.

Every integrity proof is taken following particular steps, which we have categorized in the following phases (see also Figure 4):

- **Suspicious code reception**: *Assembly and Test Server* forwards to *TIP* server a compressed file with the *suspicious* source code. If the *uncompressing* phase is not successful, this file is discarded and the integrity proof is considered invalid.

- **Trust code reception**: *TIP* server retrieves source code from the Git repository that is considered as trusted.
- **BigHashes proofs**: *TIP* server verifies, using the respective *TPM* functionalities, that the content of the compress file and the corresponding source code from the repository are identical. This is conducted consulting every hash file from the Git server. These Git registered metadata are linked as a unique chain named bigHash and the *TPM* hash functions are used to verify bigHash values. Therefore, when both bigHash values (project bigHash and compressed file's bigHash) are identical, integrity proof is considered successful.

4.2. Technical Approach and Methodology

In this section we analyze how *P2ISE* internally relies on the aforementioned processes to achieve the 3-factor integrity validation proof.

4.2.1. First Integrity Validation Check

For the first integrity check, we assume that the *Developer* has already executed Git commands to the *Source Code Control Server*. The latter forwards the changes to the *TIP*. Hence, a temporary folder within the *TIP* Server is created to contain every Jenkins work-space file. Next, all files from Jenkins work-space are compressed into a file (e.g., ZIP file), and the first security check is initiated. Once the compression is completed, the files from the selected folder are taken and filtered, and those included in the *ToExclude* list are removed preserving work-space. Once the file is sent to the *TIP* server, we have a variable $tipServer as a script input parameter. *TIP* server is implemented in PHP and it contains the *gateway.php* file which is the main responsible for the TIP server and includes the configurable variables. Most of those variables are HTTP control headers to allow remote deployment of the *TIP* server. Once all settings are done, then the *TIP* server tries to decompress it. If the decompression process is successfully completed, then the first integrity validation check has been concluded. Moreover, the file is uncompressed in a folder labeled as *suspect*. We have to note that the communication among the *Source Code Control Server* repository and the *TIP* server are performed through *POST* requests.

4.2.2. Second Integrity Validation Check

During the second integrity check, the trustworthy repository cloning takes place. Once it is successfully cloned, the BigHash values are computed using a PowerShell script and then *TPM* hashes are retrieved from the trusted repository. After the BigHash value of *suspicious* repository has been also computed, the two bigHash values are compared and the result of validity is obtained.

In our scheme, to compute the hash values, we use the SHA-256 function taking into account both the level of security provided (SHA-256 is considered secure, while for SHA-1 several vulnerabilities have been identified [47]), as well as the length of the output. Specifically, the *P2ISE* solution takes advantage of the available *TPM* functions to compute the hash values, so the size of the hash would not exceed the *TPM* input buffer limit, which is 32 bytes [48]. Moreover, to compute a complete Git folder hash, every file has to be accessed to link every hash value to a file.

4.2.3. Third Integrity Validation Check

The third integrity validation check completely relies on the intrinsic functionality of the *TPM*. *TPM* equipped computers can use the *TPM* functions for issuing and using keys that never leave the chip. These keys are used by internal functions within the chip and can only be accessed by authorized interfaces, but keys are never accessible. This fact enables the protection of created key from disclosure. *TPM* works with a particular key hierarchy that starts with an endorsement root key that is unique for each *TPM* chipset and is assigned while manufacturing. We highlight that the private part of the endorsement key will not be exposed as we have used it in the TIP server.

This step consists of each change being submitted to the Git server carrying a complete copy of the project being signed using the *Developer's* private key. This key is considered as *trusted* since it is created and stored within the *TPM*. To this end, the TIP server stores the project copy when integrity proof is required; it can be decrypted using *Developer's* public key (see Figure 5). Therefore, three copies are taken as input integration proofs, these versions should be identical. Creating a private key inside the *TPM* is a trivial process while extracting this key to a hard disk is not. At this point, we have to mention that we have taken into consideration the fact that *CI/CD* ecosystems include users with different privileges. This, however, does not create any problems to the proposed *P2ISE* solution, as the user who uploads the code can also upload updates without corrupting any step imposed by the *CI/CD* process.

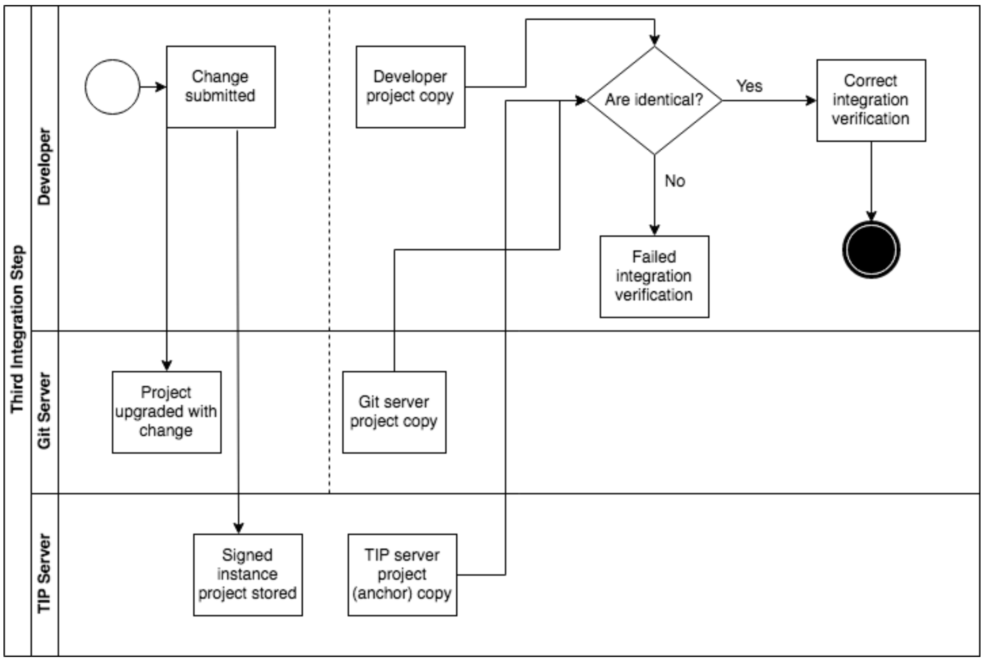
Third Integration Step
Developer
Git Server
TIP Server
Change submitted
Developer project copy
Are identical?
Yes
No
Correct integration verification
Failed integration verification
Project upgraded with change
Git server project copy
Signed instance project stored
TIP server project (anchor) copy

Figure 5. Third integration verification step—developer verification.

4.3. *Security Appraisal*

In this Section, we evaluate *P2ISE* against the nine threats that have been identified (see Section 2) using the STRIDE methodology. The proposed tool effectively addresses all threats.

First, Elevation using impersonation, from *Source Code Control Service* (*T*1), *Assembly and Test Server* (*T*2) as well as from *Public Service* (*T*8) can be prevented by *P2ISE* since it takes advantage of the TIP approach. The TIP server is equipped with a controlled server stack that guarantees that there is no possibility of containing malicious code, which leads to the fact that the software project is protected against any possible threat of impersonation. Moreover, *P2ISE* can successfully avert threats related to memory tampering of *Source Code Control Service* (*T*4) and *Assembly and Test Server* (*T*6). Again, this is achieved thanks to the use of *TPM* that it is a tamper-proof device. This valuable feature has been extensively presented in Section 3 and it is the main reason for choosing to integrate the *TPM* in *P2ISE* instead of a TEE. Last but not least, the design of the proposed solution can avert attacks related to the implementation of a weak authentication scheme (*T*3). As we have already mentioned in the above paragraphs, *P2ISE* utilizes the sealed bind keys of the *TPM* device. Finally, as mentioned in the description of the protocol, communications among the participated entities are designed to avoid possible collusion (*T*5), replay (*T*7), and cross-site scripting attacks (*T*9). Concluding, we can observe that the presence of a *TPM* and a trusted boot system that guarantees each of the boot and execution steps is necessary.

5. Performance Evaluation

In this section, we analyze the performance of the proposed tool investigating its feasibility and efficiency. We specifically focus on the added overhead as a result of the newly introduced *TIP* to different *CI/CD* processes. Due to the nature of the integrity checking process, the results vary from project to project and are also highly depended on hardware performance. For this reason, several different tests have been performed and the results offer, in conclusion, a baseline reference for the evaluation of the proposed scheme using relatively modern and fast x86 hardware.

For the prototype implementation, we developed *P2ISE* in C# language utilizing the *TPM* library that is also written in C# [49]. Also, the *TIP* server has been implemented using PowerShell scripts and receives a PHP script as input. Moreover, Powershell 7.2 was chosen as the CLI to be used for the communication among Jenkins and the *TIP* server. For the prototype evaluation, we have employed a desktop PC equipped with an AMD Ryzen 2700 CPU at 3.7 GHz, 32GB RAM, and an AMD *TPM* v3.6.0.3 (compliant with the *TPM* 2.0 specification) integrated into the ASUS ROG B450-F motherboard. Regarding the software that was used to perform these benchmarks, the PC's OS was Microsoft Windows 10 Pro 20H2, and Jenkins was the *CI/CD* environment of choice. The reason behind designing and developing *P2ISE* for Windows OS is that many large organizations have been utilizing Windows Servers to run their services. Besides, Microsoft Server was the market leader with a 48% share of the total server OS shipments in 2018 [50]. *P2ISE* is a tool, which can be integrated into day-to-day processes by these organizations that rely on the code integrity of their projects providing strong integrity guarantees.

To assess the performance of *P2ISE*, we calculated the median duration time of each process individually: (i) Integrity check; (ii) *CI/CD* build process, and (iii) Dependency tree resolution. For evaluation purposes, we decided to assess the performance of our tool against three well-known and open-source projects: (i) the Caddy Server v2 project [51] which has been developed in Go language and is approximately 32k lines of code (LoC); (ii) the Nuxt.js+Vuetify project [52], developed in JavaScript with around 1427k LoC; (iii) the Svelte project [53] developed primarily in JavaScript with only 318 LoC. In all the above cases, the LoC is counted using scc (scc. Sloc, Cloc and Code on GitHub. Retrieved 29 March 2021, from https://github.com/boyter/scc/ (accessed on 22 August 2021). These three projects with different LoC were selected to highlight how *P2ISE* affects the deployment of projects based on their LoC. While large software projects are more common to come across compared to small ones like Svelte, through these tests we aimed to assess the proposed solution's performance against both types of projects. To calculate the median duration of each process we executed each experiment 5 times and only the necessary processes were being executed at the same time. Jenkins and the TIP server were managed through a web browser application software and only one Jenkins job ran at a time to prevent possible hardware bottleneck. The duration of each process was measured via the Jenkins timestamp plugin. The results are as follows (see also Table 3):

Caddy Server v2: Measuring the performance of our tool against this project, we noticed that the integrity check overhead does not exceed the project's compiling time. The integrity check overhead was found to be stable between the different tests performed and the results show that *P2ISE* adds a small delay compared to the advantages it bears by ensuring the software integrity.

Nuxt.js+Vuetify project: Assessing the performance of our tool against this project, we highlight that the integrity check overhead has been consistent throughout the testing process and practically negligible.

Svelte project: For Svelte, we observe that the integrity check overhead is on par with the build time, which we consider a reasonable addition to this project since the overall time spent on each *CI/CD* cycle is very low. Finally, it is interesting to mention here that the Svelte project is compiled even faster than other projects with similar LoC, because it is acting as a compiler itself.

From the numerical results we can deduce that the overhead caused by our tool is little and well within reason. Overall, it is beyond any doubt that the development community will greatly benefit from adopting the proposed technique since it creates a much safer *CI/CD* pipeline.

Table 3. Median duration per process.

Process Duration (in Seconds)	Caddy Server v2	Nuxt + Vuetify Server	Svelte Server
Integrity check	5.5	2.2	1.9
CI/CD build process	7.1	36.6	1.7
Dependency tree resolution	n/a	13.6	2.6
Baseline	1.1	39.12	2.1

Figure 6 depicts the summarized results of the performance evaluation for each process. The obtained results have been compared in terms of total lines of code of each project, establishing homogenization between them. As expected, the heaviest process is the build process while the time required for the integrity check, regardless of the number of lines, remains low and stable. We consider this as an indisputable advantage of our tool, since the newly introduced integrity check process does not severely affect the overall developer routine.

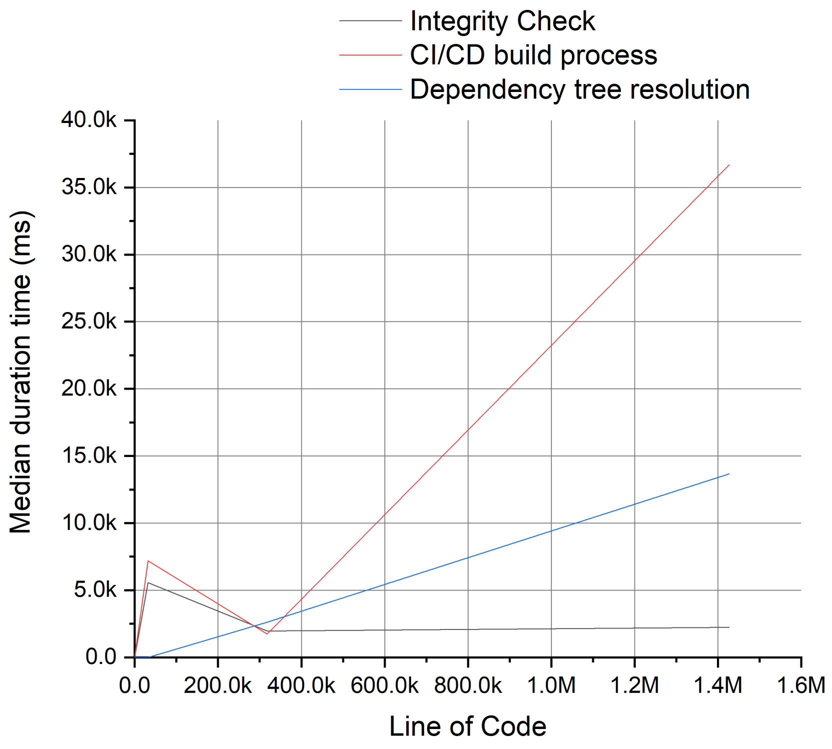

Figure 6. Performance evaluation results.

Additionally, we measured the average CPU utilization and memory consumption of the processes (Integrity check, *CI/CD* build process and the dependency tree resolution) as shown in Table 4. Regarding the *Caddy Server v2*, we observed that the CPU utilization for the integrity check is 10.6% while for the *CI/CD* build process is 38.3%; the memory consumption is 26.4% for both of said processes. As we mentioned above, we did not evaluate the process of dependency tree resolution since it is supported by this project. However, for the Nuxt+Vuetify and Svelte Server projects, we computed the CPU utilization and memory consumption for all processes. The integrity check process for the Nuxt+Vuetify Server project was 6.5% and the memory consumption was 23.2%; the *CI/CD* building process used the 10.2% of the CPU and the 26.1% of the memory, while the dependency tree resolution process utilized the 26.6% of CPU and the 23.6% of memory. Also, the CPU utilization for the Svelte Server during the integrity check, *CI/CD* build process and the dependency tree resolution was 14.8%, 14.2%, and 25% respectively, while the memory consumption for the aforementioned processes was 25.7%, 26%, and 25.9%. Overall, the most resource-consuming process is apparently the *CI/CD* build while the integrity check that is introduced by *P2ISE* is usually well below the consumption percentages recorded for the build process, regardless of the project and its size (LoC). Last but not least, we can observe that the introduction of a *TPM* chip does not entail a high additional cost. Finally, the results of our experiments have confirmed that the *P2ISE* processes do not deplete developers' resources neither delay the total deployment time, while they guarantee that the final product maintains the integrity of the source code.

Table 4. P2ISE overhead.

Software Project	P2ISE Process	CPU Utilization	Memory Consumption
Caddy Server v2	Integrity Check	10.6%	26.4%
	CI/CD build process	38.3%	26.4%
	Dependency tree resolution	n/a	n/a
Nuxt + Vuetify Server	Integrity Check	6.5%	23.2%
	CI/CD build process	10.2%	26.1%
	Dependency tree resolution	26.6%	23.6%
Svelte Server	Integrity Check	14.8%	25.7%
	CI/CD build process	14.2%	26%
	Dependency tree resolution	25%	25.9%

6. Security Analysis

In this section, we evaluate the security level provided by *P2ISE* in relation with the security requirements presented in Section 2. The results show that the proposed tool meets all the objectives, a conclusion that can be further corroborated in different cases. First, even an adversary who managed to steal the credentials of a legitimate developer is not in position to manipulate the source code, since he cannot verify his identity because the keys used for this purpose (see Section 4) safely reside in the *TPM*. This way, both confidentiality and integrity are achieved.

Moreover, all modern *CI/CD* environments keep records of developers' actions, using them as evidence also in cases where an abnormal or malicious behavior is detected. *P2ISE* ensures the accountability and non-repudiation for each one of these actions (i.e., commit command) by having them signed with the developer's secret key which is securely stored in the *TPM*. This way, no participants can deny their actions since they can be uniquely identified through the use of their key.

Critical processes such as the generation and storage of cryptographic keys, and the execution of other important cryptographic functions (i.e., hush functions) take place within the *TPM* chip. Moreover, in our scheme, each developer authenticates himself by utilizing the *TPM*'s unique key, that is hardcoded into itself. Due to the above, adding a layer of physical protection becomes essential, as all security critical procedures are bound to the hardware. By leveraging the *TPM* technology which provides multiple physical security mechanisms, the *P2ISE* operations are also proofed against physical attacks. Overall, *P2ISE* takes advantage of the features that are provided by the *TPM* to improve the *CI/CD* security.

Finally, an assertion indirectly related to the security characteristics of the proposed scheme is that instead of designing new protocols from scratch, we have opted for a solution that includes a long-established technology. More specifically, *P2ISE* is based on a solution that has been extensively analyzed and reviewed, and up to now, there are no imminent threats that can break its security properties. This makes *P2ISE* not only provably secure but also easier to be incorporated into industrial development environments.

7. Conclusions

This paper is the first to introduce an integrity preserving tool, specifically designed for developers that use *CI/CD* pipelines to manage their software projects. As the security status of a project depends not only on the underlying IT infrastructure, but also on the personal security habits of the *Developers*, it inherits the security considerations of both. Based on this observation, in this paper we proposed, designed, and implemented the *P2ISE*, a novel integrity preserving tool for *CI/CD* pipelines based on the use of secure elements. The crux of *P2ISE* is the *TPM* trusted technology, which offers undeniable integrity assertions in the project and helps prevent unauthorized actions. Having designed and implemented the *P2ISE*, we quantitatively evaluated its performance and showed that it can cope with highly demanding projects without depleting developers' resources. As the number of *Developers* who leverage the *CI/CD* pipelines in their software delivery routine is expected to increase over time, new security challenges will emerge. We hope that the research outcomes of this work become a precursor for designing schemes, frameworks, and tools for enhancing the security features of the *CI/CD* pipelines, as we did with the newly introduced *P2ISE*.

The research outcomes of this paper can be extended as future work in many ways. For this proof-of-concept implementation of *P2ISE*, we designed and developed a prototype for Windows environments. Next, we plan to implement *P2ISE* for Linux and Unix-based servers, use it alongside different *Assembly and Test Servers* environments besides Jenkins and GitLab, and finally test its performance against large-scale software projects and distributed development environments. This will help us identify additional use-cases for our tool, optimize its existing features, and extend its functionality with new ones.

Author Contributions: Conceptualization, A.M., A.F. and J.R.C.C.; methodology, A.M.; software, A.M. and J.R.C.C.; validation, A.M., A.F., J.R.C.C. and C.X.; formal analysis, A.M., A.F. and C.X.; investigation, A.M. and A.F.; resources, A.M.; data curation, A.M. and A.F.; writing—original draft preparation, A.M., A.F. and C.X.; writing—review and editing, A.M. and A.F.; visualization, A.M. and A.F.; supervision, C.X.; project administration, A.M. and A.F.; funding acquisition, C.X. All authors have read and agreed to the published version of the manuscript.

Funding: The present work benefits from the financial support of the European Commission, under the SealedGRID, SECONDO and CUREX projects; Grant Agreements Nos. 777996, 823997 and 826404, respectively.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bass, L.; Weber, I.; Zhu, L. *DevOps: A Software Architect's Perspective*; SEI Series in Software Engineering; Addison-Wesley: New York, NY, USA, 2015.
2. Humble, J.; Farley, D.G. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*; Addison-Wesley: Upper Saddle River, NJ, USA, 2010.
3. Tichy, M.; Goedicke, M.; Bosch, J.; Fitzgerald, B. Rapid Continuous Software Engineering. *J. Syst. Softw.* **2017**, *133*, 159. [CrossRef]
4. DigitalOcean. CURRENTS: A Quarterly Report on Developer Trends in the Cloud. Available online: https://assets.digitalocean.com/currents-report/DigitalOcean-Currents-Q1-2018.pdf (accessed on 22 August 2021).
5. André, P.; Cardin, O. Trusted services for cyber manufacturing systems. In *Service Orientation in Holonic and Multi-Agent Manufacturing*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 359–370.
6. Ellingwood, J. An Introduction to CI/CD Best Practices. 2018. Available online: https://www.digitalocean.com/community/tutorials/an-introduction-to-ci-cd-best-practices (accessed on 29 June 2021).
7. National Cyber Security Centre. Multi-Factor Authentication for Online Services. 2013. Available online: https://www.ncsc.gov.uk/guidance/multi-factor-authentication-online-services (accessed on 29 June 2021).
8. Milka, G. Anatomy of Account Takeover. Available online: https://2018.swisscyberstorm.com/wp-content/uploads/2018/11/The-Anatomy-of-Account-Takeover.pdf (accessed on 29 June 2021).

9. Paule, C.; Düllmann, T.F.; Van Hoorn, A. Vulnerabilities in Continuous Delivery Pipelines? A Case Study. In Proceedings of the 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, 25–26 March 2019; pp. 102–108.
10. Sathyanarayanan, N.; Nanda, M.N. Two Layer Cloud Security Set Architecture On Hypervisor. In Proceedings of the 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC), Bangalore, India, 9–10 February 2018; pp. 1–5.
11. Mahboob, J.; Coffman, J. A Kubernetes CI/CD Pipeline with Asylo as a Trusted Execution Environment Abstraction Framework. In Proceedings of the 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC), Online, Virtual Conference, Las Vegas, NV, USA, 27–30 January 2021; pp. 0529–0535.
12. Rangnau, T.; Buijtenen, R.V.; Fransen, F.; Turkmen, F. Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. In Proceedings of the 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC), Eindhoven, The Netherlands, 5–8 October 2020; pp. 145–154. [CrossRef]
13. Bass, L.; Holz, R.; Rimba, P.; Tran, A.B.; Zhu, L. Securing a deployment pipeline. In Proceedings of the 2015 IEEE/ACM 3rd International Workshop on Release Engineering, Florence, Italy, 19 May 2015; pp. 4–7.
14. Ullah, F.; Raft, A.J.; Shahin, M.; Zahedi, M.; Babar, M.A. Security support in continuous deployment pipeline. *arXiv* **2017**, arXiv:1703.04277.
15. Rimba, P.; Zhu, L.; Bass, L.; Kuz, I.; Reeves, S. Composing patterns to construct secure systems. In Proceedings of the 2015 11th European Dependable Computing Conference (EDCC), Paris, France, 7–11 September 2015; pp. 213–224.
16. Muñoz, A.; Farao, A.; Correia, J.R.C.; Xenakis, C. ICITPM: Integrity validation of software in iterative Continuous Integration through the use of Trusted Platform Module (TPM). In *European Symposium on Research in Computer Security*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 147–165.
17. Buchanan, S.; Rangama, J.; Bellavance, N. CI/CD with Azure Kubernetes Service. In *Introducing Azure Kubernetes Service*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 191–219.
18. Mohan, V.; Othmane, L.B. Secdevops: Is it a marketing buzzword?-mapping research on security in devops. In Proceedings of the 2016 11th International Conference on Availability, Reliability and Security (ARES), Salzburg, Austria, 31 August–2 September 2016; pp. 542–547.
19. Jauernig, P.; Sadeghi, A.R.; Stapf, E. Trusted execution environments: properties, applications, and challenges. *IEEE Secur. Priv.* **2020**, *18*, 56–60. [CrossRef]
20. Moon, S.J.; Sekar, V.; Reiter, M.K. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In Proceedings of the 22nd Acm Sigsac Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 1595–1606.
21. Deepa, G.; Thilagam, P.S. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Inf. Softw. Technol.* **2016**, *74*, 160–180. [CrossRef]
22. Lee, T.; Won, G.; Cho, S.; Park, N.; Won, D. Detection and mitigation of web application vulnerabilities based on security testing. In Proceedings of the IFIP International Conference on Network and Parallel Computing, Gwangju, Korea, 6–8 September 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 138–144.
23. Lipke, S. Building a Secure Software Supply Chain. 2017. Available online: https://hdms.bsz-bw.de/files/6321/20170830_thesis_final.pdf (accessed on 29 June 2021).
24. Schneider, C. Security DevOps-Staying Secure in Agile Projects. Available online: https://docplayer.net/19676868-Security-devops-staying-secure-in-agile-projects-christian-schneider-cschneider4711.html (accessed on 29 June 2021).
25. Koutroumpouchos, N.; Ntantogian, C.; Xenakis, C. Building Trust for Smart Connected Devices: The Challenges and Pitfalls of TrustZone. *Sensors* **2021**, *21*, 520. [CrossRef] [PubMed]
26. Cerdeira, D.; Santos, N.; Fonseca, P.; Pinto, S. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1416–1432.
27. Farao, A.; Rubio, J.E.; Alcaraz, C.; Ntantogian, C.; Xenakis, C.; Lopez, J. SealedGRID: A Secure Interconnection of Technologies for Smart Grid Applications. In *Critical Information Infrastructures Security*; Nadjm-Tehrani, S., Ed.; Springer International Publishing: Cham, Switzerland, 2020; pp. 169–175.
28. Farao, A.; Veroni, E.; Ntantogian, C.; Xenakis, C. P4G2Go: A Privacy-Preserving Scheme for Roaming Energy Consumers of the Smart Grid-to-Go. *Sensors* **2021**, *21*, 2686. [CrossRef] [PubMed]
29. Matetic, S.; Schneider, M.; Miller, A.; Juels, A.; Capkun, S. DelegaTEE: Brokered delegation using trusted execution environments. In Proceedings of the 27th USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 1387–1403.
30. Vasudevan, A.; Owusu, E.; Zhou, Z.; Newsome, J.; McCune, J.M. Trustworthy Execution on Mobile Devices: What Security Properties can My Mobile Platform Give Me? Available online: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.220.220&rep=rep1&type=pdf (accessed on 29 June 2021).
31. Beniamini, G. War of the Worlds-Hijacking the Linux Kernel from QSEE. Available online: http://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html (accessed on 29 June 2021).

32. Machiry, A.; Gustafson, E.; Spensky, C.; Salls, C.; Stephens, N.; Wang, R.; Bianchi, A.; Choe, Y.R.; Kruegel, C.; Vigna, G. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In Proceedings of the NDSS'17, San Diego, CA, USA, 26 February–1 March 2017.
33. Rosenberg, D. Reflections on Trusting Trustzone. Available online: https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2014/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf (accessed on 29 June 2021)
34. Chen, Y.; Zhang, Y.; Wang, Z.; Wei, T. Downgrade attack on trustzone. *arXiv* **2017**, arXiv:1707.05082.
35. Osvik, D.A.; Shamir, A.; Tromer, E. Cache attacks and countermeasures: The case of AES. In Proceedings of the Cryptographers' Track at the RSA Conference, San Jose, CA, USA, 13–17 February 2006; pp. 1–20.
36. Yarom, Y.; Falkner, K. FLUSH + RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 719–732.
37. Gruss, D.; Maurice, C.; Wagner, K.; Mangard, S. Flush + Flush: A fast and stealthy cache attack. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, San Sebastián, Spain, 7–8 July 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 279–299.
38. Brasser, F.; Müller, U.; Dmitrienko, A.; Kostiainen, K.; Capkun, S.; Sadeghi, A.R. Software grand exposure: SGX cache attacks are practical. In Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT 17), Vancouver, BC, Canada, 14–15 August 2017.
39. Moghimi, A.; Irazoqui, G.; Eisenbarth, T. Cachezoom: How SGX amplifies the power of cache attacks. In Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems, Taipei, Taiwan, 25–28 September 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 69–90.
40. Schwarz, M.; Weiser, S.; Gruss, D.; Maurice, C.; Mangard, S. Malware guard extension: Using SGX to conceal cache attacks. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Taipei, Taiwan, 25–28 September 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 3–24.
41. Lipp, M.; Gruss, D.; Spreitzer, R.; Maurice, C.; Mangard, S. Armageddon: Cache attacks on mobile devices. In Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), Austin, TX, USA, 10–12 August 2016; pp. 549–564.
42. Zhang, N.; Sun, K.; Shands, D.; Lou, W.; Hou, Y.T. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptol. ePrint Arch.* **2016**, *2016*, 980.
43. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Mangard, S.; Kocher, P.; Genkin, D.; Yarom, Y.; Hamburg, M. Meltdown. *arXiv* **2018**, arXiv:1801.01207.
44. Kocher, P.; Horn, J.; Fogh, A.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; et al. Spectre attacks: Exploiting speculative execution. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 1–19.
45. Dheerendra, M.; Sourav, M.; Saru, K.; Khurram, K.M.; Ankita, C. Security enhancement of a biometric based authentication scheme for telecare medicine information systems with nonce. *J. Med. Syst.* **2014**, *38*, 41.
46. Saru, K.; Kumar, D.A.; Xiong, L.; Fan, W.; Khurram, K.M.; Qi, J.; Hafizul, I.S. A provably secure biometrics-based authenticated key agreement scheme for multi-server environments. *Multimed. Tools Appl.* **2018**, *77*, 2359–2389.
47. Stevens, M.; Bursztein, E.; Karpman, P.; Albertini, A.; Markov, Y. The first collision for full SHA-1. In Proceedings of the Annual International Cryptology Conference, Santa Barbara, CA, USA, 20–24 August 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 570–596.
48. IBM's TPM 2.0 TSS. Available online: https://sourceforge.net/projects/ibmtpm20tss/ (accessed on 29 June 2021).
49. Microsoft. TPM Software Stack (TSS) Implementations from Microsoft. 2013. Available online: https://github.com/microsoft/TSS.MSR (accessed on 9 August 2021).
50. Server Operating System Market Share. Available online: https://www.t4.ai/industry/server-operating-system-market-share (accessed on 5 August 2021).
51. Caddy Server v2 Official One. Available online: https://github.com/caddyserver/caddy (accessed on 29 June 2021).
52. Nuxt.js + Vuetify Project. Available online: https://gitlab.com/tip-benchmarking/nuxt-vuetify (accessed on 29 June 2021).
53. Svelte Project. Available online: https://gitlab.com/tip-benchmarking/svelte (accessed on 29 June 2021).