Software Supply-Chain Security: Issues and Countermeasures

Badis Hammi, EPITA
Sherali Zeadally, University of Kentucky

Software application development involves various actors and organizations in what is called the software supply chain. *We discuss how we can achieve strong resilience of the software supply chain to cyberthreats and then propose a holistic end-to-end security approach for the software supply chain.*

Software application development is a complex activity that involves various actors and organizations in what is called *the software supply chain*. The evolution of the software supply chain led to numerous benefits, such as profit maximization, code mutualization, and the optimization of lead times. However, the complexity of the software supply chain results in multiple security issues and attacks because compromises are highly prevalent. An attacker that compromises a single link (for example, by maliciously modifying the software) in the software supply chain, can harm users of this software and this attack technique is frequently being exploited to attack high-profile companies. We can provide a holistic and effective security solution to the software supply chain only if its security state and features are well understood.

INTRODUCTION

A supply chain is a global network that delivers raw materials, products, and services to end customers through an engineered flow of information, physical

Digital Object Identifier 10.1109/MC.2023.3273491
Date of current version: 26 June 2023

0018-9162/23

distribution, and money. Figure 1 illustrates a basic supply chain with three entities: a supplier, one producer, and one customer. Four basic flows connect these entities together: 1) a flow of physical materials and services (materials, components, supplies, services, and finished products) from the supplier to the end customer, 2) a flow of cash from the end customer to the raw material supplier, 3) a flow of information (invoices, sales literature, specifications, receipts, orders, and rules and regulations) back and forth along the chain, and 4) a reverse flow of products returned (returns for repair, replacements, recycling, and disposals).

The rapid growth of information communication technologies (ICT) has impacted many fields. In this context, the supply chain has also quickly evolved toward the digital supply chain (DSC), where digital and electronic technologies have been integrated into every aspect of the end-to-end supply chain. These technologies are radically transforming supply-chain structures in different sectors, which have resulted in multiple benefits, such as increased profit and reduced loss, the optimization of supply-chain lead times, the reduction of markdowns and stock-outs, and improved collaborations among different stakeholders.[1] However, the DSC is vulnerable to a wide range of cyberattacks that can range from simple information theft to complete stoppage of a factory's activities. Indeed, the DSC does not rely on a single technology, but on the integration of different technologies, such as the Internet of Things, cloud computing, networks and telecommunications, and many others. Thus, the DSC is vulnerable to the various cyber risks associated with the underlying technologies.[1]

The discipline that addresses cybersecurity risks that are related to extended supply chains and supply ecosystems is known as *cyber supply-chain risk management* (*C-SCRM*).[2] C-SCRM broadly comprises concepts, such as third-party risk management and external dependency management.[2] According to Boyson,[3] C-SCRM is an overarching discipline that combines cybersecurity, enterprise risk management, and supply-chain management into a new and powerful concept that provides strategic control over the end-to-end processes of an organization and its extended partners. Therefore, currently, more than ever before, supply-chain management and C-SCRM must be an integral part of a business and are vital to any company's success.[4]

THE EVOLUTION OF THE SOFTWARE SUPPLY CHAIN LED TO NUMEROUS BENEFITS, SUCH AS PROFIT MAXIMIZATION, CODE MUTUALIZATION, AND THE OPTIMIZATION OF LEAD TIMES.

For any physical product, Figure 1 shows the typical supply-chain model followed. However, for software supply chains in particular, the model is slightly different. As Figure 2 depicts, all of the steps (also called *links*) of the software supply chain are like the steps of the traditional supply chain.

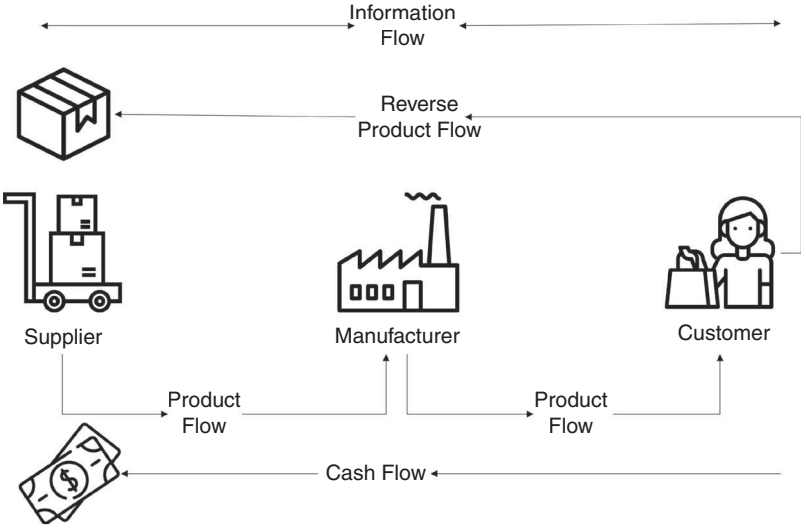

FIGURE 1. A basic supply chain.

In the highly connected world today, most organizations depend on other organizations for their products and services. Software producers are not an exception, and a software supply chain involves multiple, different actors throughout different organizations. Indeed, developers use source code available in version control systems (like raw materials). This source code is then compiled into binaries. The resulting software is packaged and published for distribution in the form of a final product that can be used by end users.[5]

Research contributions of this work

The central research question we aim to address in this work is: How can a software supply chain be resilient to cyberthreats? To answer this question, we need a good understanding of the current state of security in the software supply chain. In this context, our work focuses on the following aspects:

- We describe the models currently used by the software supply chain.
- We discuss the security issues and attacks that threaten the software supply chain.
- We present current protection mechanisms that can improve the security of the software supply chain.
- We propose a holistic approach and model for end-to-end security of the software supply chain.

CURRENTLY, SOFTWARE SUPPLY-CHAIN SECURITY ISSUES ARE THE FASTEST GROWING THREATS TO INTERNET USERS.

Unfortunately, like their physical counterparts, security issues are very common in the software supply chain. According to O'Gorman et al.,[6] software supply-chain security issues rose by 438% from 2017 to 2019 and by over 300% in 2021. Gartner (https://www.gartner.com/en/documents/4003625) predicts that by 2025, 45% of organizations would have experienced a software supply-chain attack. Currently, software supply-chain security issues are the fastest growing threats to Internet users. Such security issues affect all types of actors, from simple end users to big companies, such as Microsoft, Google, and Redhat.[5] Despite the continuous rise in security issues and attacks, the software supply chain and its protection mechanisms have not really received commensurate attention as does the physical supply chain.[5]

SOFTWARE SUPPLY-CHAIN MODELS

Figure 2 shows a macroscopic overview of a software supply chain. However, a microscopic view shows that a software supply chain is not always completely linear, especially for complex systems, such as operating systems. Understanding all of the details related to a supply chain is key to secure the product software and the supply chain itself. In fact, there are multiple software supply-chain models that have emerged in recent years. For example, Figure 3 describes three different software supply-chain models, mainly related to operating systems. (We choose operating systems as an example because they are among the most complex software systems.) However, there are many others[5]:

- *A software supply chain often relies on distribution packaging*, which is a very common technique. Multiple Linux operating systems, such as Debian or Fedora distributions-based systems, supply a preinstalled package manager that enables users to install packages that

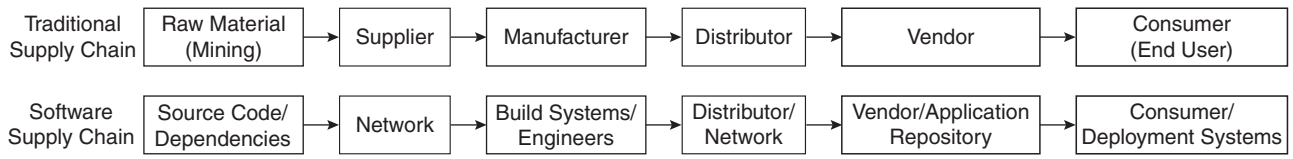

FIGURE 2. Comparison of a traditional supply chain and a software supply chain.

are selected and maintained by the operating system distribution development team (https://wiki.debian.org/Apt). A similar technique is used by Microsoft and Apple through application and software stores (https://www.apple.com/app-store/). For example, Figure 3(a) describes the steps to create a distribution package for a Debian-based software. It comprises three steps: 1) retrieving upstream sources that look at application code from their source (for example, the use of a well-known protocol code from its source), 2) application of the changes specific to the platform, and 3) the production of a platform-specific installable package (for example, .pkg or .deb) using both the upstream release and the patched and modified code.

- *A software supply chain for live media:* Generally, a live medium comprises various software components that are from different producers. All of the software components are packaged into a bootable disk image. With such a configuration, the bootable media tooling fetches artifacts and executes platform-specific configuration scripts on them.[5] Figure 3(b) describes such a model.
- *A software supply chain for application-oriented operating systems:* There exist multiple

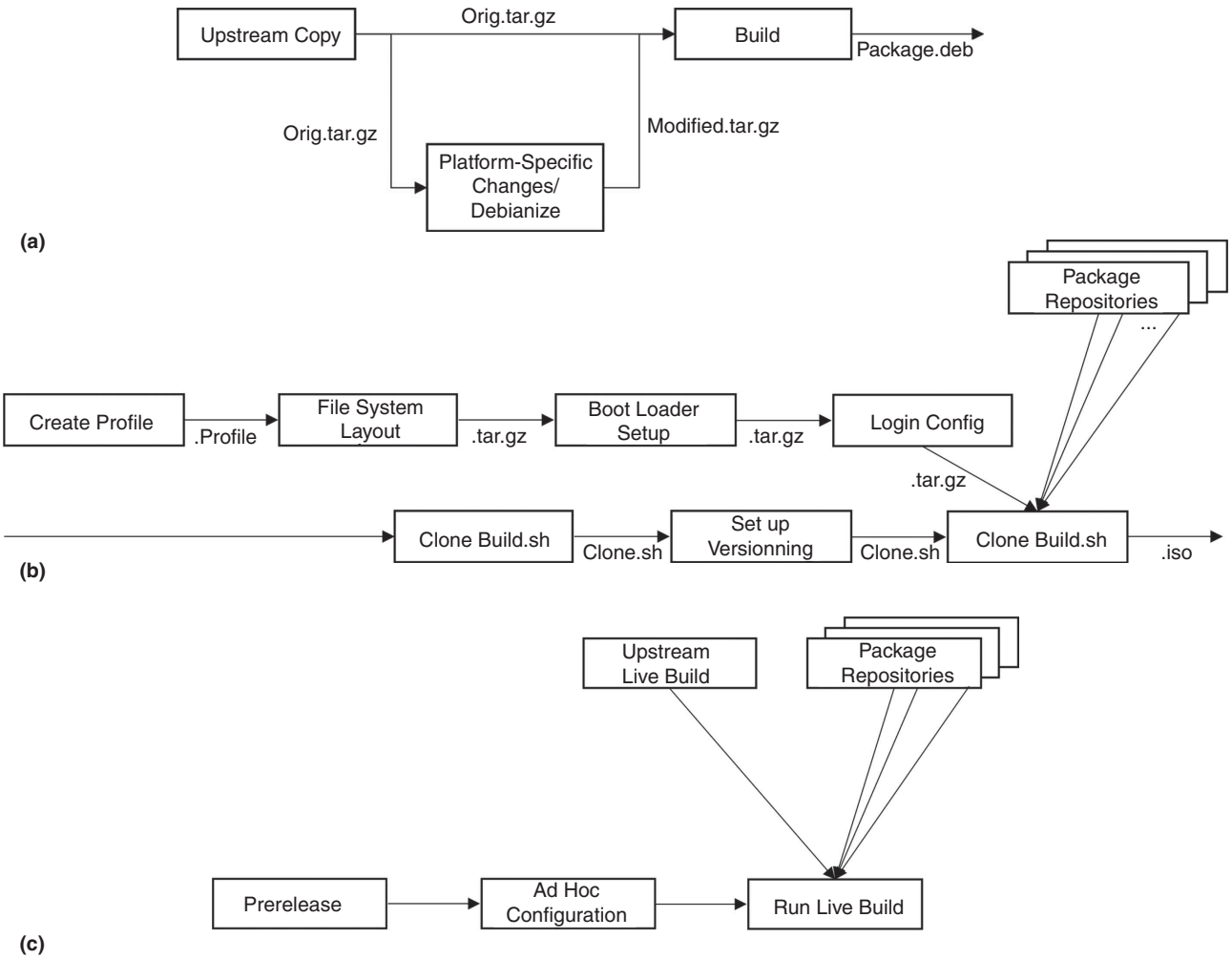

FIGURE 3. Nonlinear software supply-chain models for (a) distribution packaging, (b) live media (.iso), and (c) live-build toolbox.[5]

application-specific operating systems (for example, Kali Linux or Parrot for security, Tails for privacy, and so on). Such products rely on a complex supply chain where the software provider only manages and modifies the configuration files while the packages that compose the operating systems are all from different providers (each of these packages followed a different supply chain). Figure 3(c) presents this model.

SOFTWARE SUPPLY-CHAIN SECURITY ISSUES AND ATTACKS

Cybercrime costs organizations $2.9 million every minute, according to RiskIQ research (https://www.fortinet.com/resources/cyberglossary/cybersecurity-statistics). In an Embroker 2021 technical report,[7] the authors predicted that cybercrime will cost companies worldwide an estimated $10.5 trillion annually by 2025. In this context, software supply-chain attacks are on the rise because attackers can attack or infiltrate large organizations and their software through a single, third-party software product. There are two entry points for attackers: 1) through intended taints, which an attacker embeds into the source code mainly through updates (for example, the SolarWind attack[8]); and 2) through software and protocol breaches and flaws discovered by the community [for example, Log4Shell common vulnerability and exposure (CVE)-2021-44228].

The concept of a software supply chain is new. Previously, the software lifecycle was considered as a set of disjointed operations. This misconception is the cause of various security issues and compromises. These compromises stem mainly from two misconceptions[5]: 1) the lack of understanding of a global view of the software lifecycle because of which security resources may not be allocated appropriately; and 2) the assumption that securing each individual phase of the software development lifecycle will provide a secure software. However, the latter results in a lack of effort in securing the transitions between these steps. More precisely, even if the security of each link and step is crucial, the end-to-end security can be compromised if attackers can tamper with the output of a step or a link before it is provided to the next one in the chain.

Software defect is often quantified according to the number of taints per 1,000 lines of code.[9] Systems have generally tens of millions of code lines (for example, Microsoft Windows operating systems have 50 million lines of code). Therefore, they represent thousands of possible vulnerabilities.

Accidental (nondeliberate) software vulnerabilities embedded into products during their design or implementation are called *unintended taints*. Such vulnerabilities are continually discovered, made public, and remediated using different types of patches. However, some systems are not updated/patched quickly or not patched at all, which make them vulnerable to various types of attacks. According to Executive Order 13800 (https://www.cisa.gov/executive-order-strengthening-cybersecurity-federal-networks-and-critical-infrastructure), known but unmitigated vulnerabilities are among the highest cybersecurity risks. One of the well-known incidents related to such security issues is the Heartbleed vulnerability[10] CVE-2014-0160, which was a security vulnerability in the OpenSSL (secure socket layer) cryptography library that allowed attackers to get access to confidential data, such as unencrypted exchanges between transport layer security (TLS) parties, authentication secrets (such as session credentials, private keys, cookies), and so on, which enable attackers to decrypt communications of compromised parties. After an attacker has gained authentication credentials, the attacker can impersonate the victim even after a security patch of Heartbleed has been applied. In other words, the attacker can impersonate the victim if the victim's credentials are still valid (for example, before changing credentials or the revocation of the private key). When the Heartbleed SSL/TLS vulnerability was announced, more than 80,000 SSL certificates were revoked in the week following the publication.[11] The unintended taint was embedded into the OpenSSL library in 2012 and was publicly revealed in April 2014.

However, system administrators are known to be generally slow in patching their systems. For example, on 20 May 2014, 1.5% of the 800,000 most popular websites that use TLS were still vulnerable to Heartbleed (https://www.theregister.com/2014/05/20/heartbleed_still_prevalent/). On 23 January 2017, according to Shodan (https://www.shodan.io/), nearly 199,594 devices connected to the Internet were still vulnerable. Another vulnerability with the same potential is Shellshock CVE-2014-6271, which is a security vulnerability in the Unix Bash shell that enables an attacker to cause Bash to execute arbitrary commands and gain unauthorized access to Internet-facing services (for example, web servers) that use Bash to process requests.[12] A few days after the publication of Shellshock, various related vulnerabilities were discovered

(CVE-2014-6277, CVE-2014-6278, CVE-2014-7169, CVE-2014-7186, and CVE-2014-7187). A few hours after the initial disclosure of Shellshock, adversaries exploited it to create botnets to perform distributed denial of service attacks and vulnerability scanning (https://www.wired.com/2014/09/hackers-already-using-shellshock-bug-create-botnets-ddos-attacks/). Also, in the few days after the initial disclosure, there were millions of scans and cyberattacks related to Shellshock (https://bits.blogs.nytimes.com/2014/09/26/companies-rush-to-fix-shellshock-software-bug-as-hackers-launch-thousands-of-attacks/).

The last wide-scale vulnerability of this type is Log4Shell (https://logging.apache.org/log4j/2.x/security.html). *Log4Shell* is the name given to the vulnerability discovered in the library Log4j, which Apache provides. It allows developers to insert log statements in JavaServer Pages without using Java scripting. According to Checkpoint (https://blog.checkpoint.com/2021/12/13/the-numbers-behind-a-cyber-pandemic-detailed-dive/), Log4j is clearly one of the most serious vulnerabilities on the Internet in recent years, with a strong potential for a huge impact. The main vulnerability described in CVE-2021-44228 is of critical severity because it allows an attacker to execute a reverse shell on the vulnerable machine with high privileges. The attacker therefore can do whatever he/she wants (for example, uploading a ransomware). Moreover, since its discovery, multiple variants have been revealed. Checkpoint researchers discovered 60 variants of this vulnerability only 24 hours after the vulnerability was disclosed, each with a different severity level (for example, the vulnerability described in CVE-2021-45046 enables a denial of service on the vulnerable machine). Log4j is a brick (one of the components) in the supply chain of numerous software and a vulnerability at its level can affect a wide range of software products, some of which are used in critical systems, such as Splunk (https://www.splunk.com), various Amazon services (for example, AWS CloudHSM, Kafka, AWS Glue, and many others), Fortiguard (https://www.fortiguard.com/), MongoDB, Okta (https://www.okta.com/), and many others. According to Checkpoint, 72 hours after the initial outbreak of Log4j, the number of attack attempts reached 800,000. A few days later, they reported 4,300,000 attack attempts because of this vulnerability, with more than 46% of those attempts launched by well-known malicious groups.

Another type of taint is the *malicious taint*, which occurs when authentic components that have been previously validated have some functionality intentionally inserted into them by some adversary, which affects their safety, reliability, and security.[9] The best example to illustrate the danger behind malicious taint is the SolarWinds supply-chain attack.[4,8,13] In the SolarWinds attack, hackers gained access through Trojanized updates to SolarWinds' Orion computer monitoring and management software. Basically, a software update was exploited to install Sunburst malware in Orion, which was then installed by almost 18,000 customers. Once installed, the malware provided hackers with a back door to SolarWinds customers' systems and networks. This attack illustrates a good example of software supply-chain vulnerabilities and consequences, because instead of directly attacking the federal government or a private organization's network, hackers targeted a third-party vendor, which provides them with software. In this case, the target was the computer management software Orion, supplied by the Texas company SolarWinds. More than 33,000 companies use Orion. SolarWinds reported that 18,000 of its customers have been affected, including 425 companies of the Fortune 500 (https://fortune.com/fortune500/). The very first attack targeted FireEye systems. FireEye is a company that assists in the security

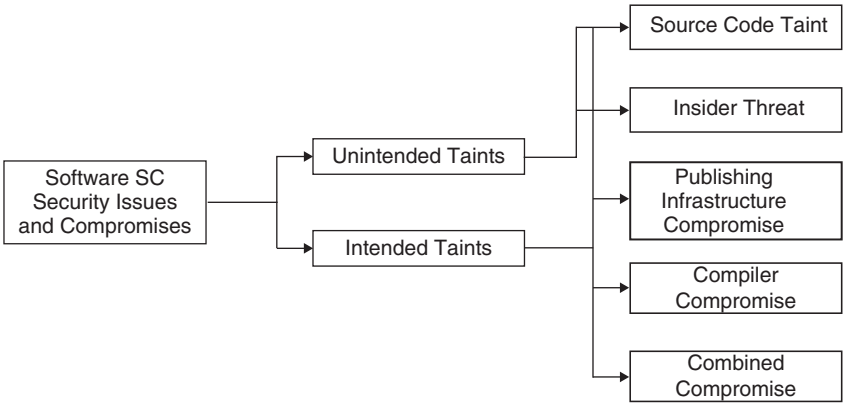

FIGURE 4. Taxonomy of security issues and attacks in the software supply chain.

management of several large private companies and federal government agencies.

Figure 4 presents a taxonomy of security issues and attacks in the software supply chain and describes multiple malicious compromises. Based on this taxonomy, we found that software supply-chain's issues are mainly due to intended taint or unintended taint. Both classes can be divided into multiple subclasses:

- *Source code taint:* This is one of the most common compromises (https://github.com/advisories/GHSA-jxf5-7x3j-8j9m; https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_Reports/2018-06-28_Github) in the software supply chain and occurs when an attacker deliberately inserts malicious code that can be exploited on target users. Such a taint is possible in two ways: 1) through software or via its updates, or 2) via an online compromise of the developers' source code (for example, an attacker who obtains the Github credentials of a developer). The SolarWinds attack we have described above is such an example.
- *Publishing infrastructure compromise:* According to Torres-Arias,[5] this is the most prevalent compromise for the software supply chain (https://securelist.com/operation-shadowhammer/89992/; https://bitcoingold.org/vulnerable-wallets/; https://blogs.windows.com/windows-insider/2017/06/01/note-unintentional-release-builds-today/; https://paper.seebug.org/papers/APT/APT_CyberCriminal_Campagin/2014/The_Monju_Incident.pdf). It represents the case where the publishing infrastructure of source code or software applications [for example, software package repository for a distribution, a community repository (like PyPI), or a project's website] gets compromised. A recent example is the case of a hacker who duped hundreds of users into downloading a version of Linux Mint with a back door. The attacker was able to build a botnet of hundreds of hosts in less than 24 hours (https://www.zdnet.com/article/hacker-hundreds-were-tricked-into-installing-linux-mint-backdoor/).
- *Insider threat:* According to the Cybersecurity and Infrastructure Security Agency,[14] insider threat is the potential for an insider (an insider is any person who has or had authorized access to or knowledge of an organization's resources, such as personnel, networks, systems, and so on) to use his/her authorized access or understanding of an organization to cause harm to that organization. Insider threat can also affect the software supply chain. Misconfigurations and/or unintended taint executed by insiders are also considered as insider threats.[14,15]
- *Developer, compiler/building tool compromises:* These represent attacks that are achieved through the backdooring of compilers, as well as developer key/account compromise. Even if they are not among the most common compromises (https://blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/; https://www.zdnet.com/article/red-hats-ceph-and-inktank-code-repositories-were-cracked/), they have substantial consequences.[5]
- *Combined multistep compromise:* Here an attacker can perform a combination of the compromises discussed above.

SECURITY SOLUTIONS FOR SOFTWARE SUPPLY CHAIN

There are numerous proposals that aim to increase the security of different aspects within the supply chain, and these include: 1) the protection of source code repository (https://dwheeler.com/; https://mikegerwitz.com/2012/05/a-git-horror-story-repository-integrity-with-signed-commits; https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work); 2) the verification of compilers, applications; and kernels[16]; and 3) package management and software distribution mechanisms.[17] However, there are very few approaches that offer a holistic, end-to-end security solution for the software supply chain.[5]

Microsoft proposed a framework that incorporates best practices of software integrity risk-management into: 1) the process of software product development, and 2) the operations of online services.[18] The framework aims to enhance the security and trustworthiness of software among the different parties (people, processes, and technologies) involved that make up a modern ICT supply chain. It follows six phases: planning, discovery, assessment, development, validation, and implementation.

Alberts et al.[19] described the Carnegie Mellon Software Engineering Institute risk assessment approach for software supply chains. The approach relies on a few factors, called *drivers*, which have a strong influence on the

eventual output or result. The experimental evaluation conducted to validate the approach showed that the development of a comprehensive profile of systemic risks to mission success requires around 15 to 25 drivers. Each driver is represented as a yes/no question, where an answer of "yes" means that the driver is in its success state. In other words, it contributes a minimal risk to the software supply-chain mission. An answer of "no" means that the driver is in its failure state. That is, it represents a severe degree of risk to the software supply-chain mission.[19]

In the same context, Simpson et al.,[20] the authors of SAFECode, developed sound assurance practices during each phase of the software development process, which can reduce the risks related to the supply chain. This approach does not design a framework or an approach as the ones (that is, Storch[18] and Alberts et al.[19]) proposed earlier in this section. But it provides a set of best practices, verifications, and controls (mainly integrity controls), during the software sourcing phase, the software development and testing phase, and the software delivery and sustainment phase.

Torres-Arias[5] proposed a framework that ensures the integrity of the supply chain as a whole by allowing actors within the software supply chain to create certifications of the actions they performed on the chain. These certifications are applied to every step in the chain and provide enough semantic information to enforce strong software supply-chain integrity and authentication checks.

SOFTWARE SUPPLY-CHAIN MODEL FOR HOLISTIC END-TO-END SECURITY

The security of the software supply chain is fundamental to the security of the final product. A taint in any link of this supply chain can lead to different types of compromise in the final software product, ranging from back doors to vulnerable libraries. Current security approaches focus on securing each link individually. These traditional approaches cannot provide a holistic end-to-end supply-chain security approach, as we have highlighted before. In this context we propose a model that can provide holistic end-to-end security of the software supply chain.

Our proposed model uses six main entities: public key infrastructures (PKI), a blockchain, developers, service providers, link agents, and final users (i.e., consumers). Figure 5(a) describes an abstraction of the software supply chain, highlighting the roles and interactions among the entities.

- A *PKI* is responsible for the distribution of certificates to the developers and service providers to ensure their authentication. Each contributor in the supply chain must be authenticated.
- A *developer* provides source code for a given supply-chain link. Like in Git, when a developer produces some code, it must sign the commit and, when a developer wants to use a code, the signature of the latter must be verified.
- A *service provider* offers remote software services after receiving a call with the execution parameters.
- A *blockchain* serves as a distributed ledger to store different types of information related to the outputs of the different phases and links in the supply chain.
- A *link agent* is a person or a process that ensures the integrity and authentication of each code used as input (received from the previous link) for a given link.
- A *final user* is a consumer of the software product.

Figure 5(b) describes an abstraction of the supply-chain model proposed. For each link, we provide an input (source code) and we produce an output noted link product (library, package, and so on). We use the following notations for our model:

- L: a given link in the blockchain.
- $H(\,)$: hash function (Merkle hash).
- $Sign(\,)$: signature algorithm (using the private key associated with the certificate).
- C_L^n: the nth code provided by the link L.
- S_L^m: the mth service provided by a service provider at the link L.
- O_L: the output of a link L (1).
- P_L: the final product of a link L (2).
- r: nonce that a decentralized authority sends to a developer who uses the nonce in multi-factor authentication.

When a developer produces some source code (product), it stores the Merkle root computed on the code in the blockchain and the transaction is signed using the developer's private key. Figure 5(c) describes how the Merkle root is computed. In the figure's example, the code noted I is produced at the link L. The code I uses 1) the codes D and E from the link $L-1$ and 2) the codes F and G are produced at the link L (added by the developer). To compute the Merkle root $H(I)$, the previous hashes (also Merkle roots) $H(D)$ and $H(E)$ are considered as nodes in the Merkle tree relative to $H(I)$, and F and G are added as new leaves, as Figure 5(c) shows. Computing such a Merkle root ensures the integrity of all the source codes used from the

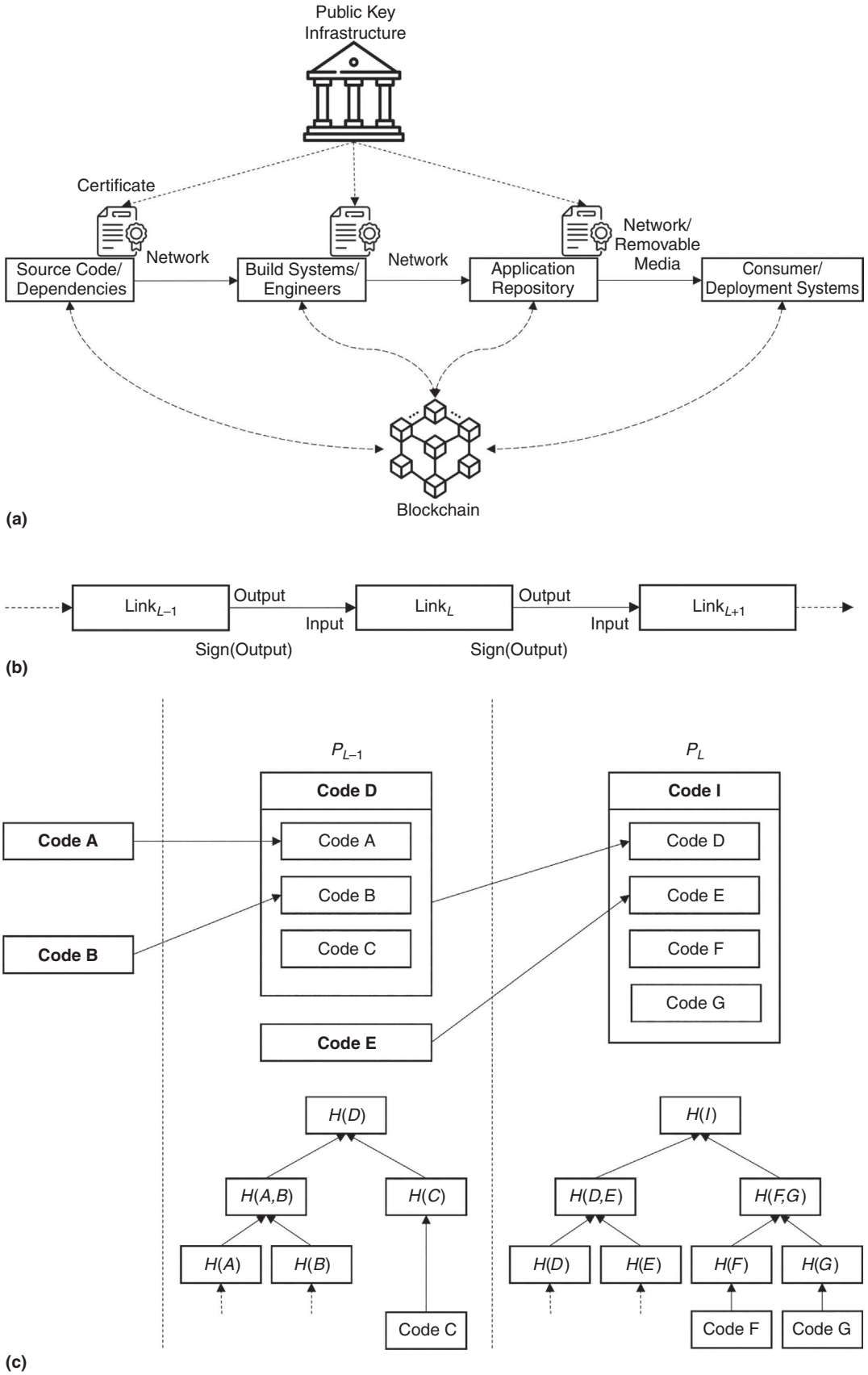

FIGURE 5. Proposed model: (a) Entities of the secure end-to-end model for the software supply chain, (b) abstraction of the software supply chain, (c) integrity of the source code along the supply chain via Merkle hashes.

previous links. Equation (1) describes the output of a link:

$$O_L := P_L \,|\, H(P_L) \,|\, \mathrm{Sign}(P_L \,|\, H(P_L)). \quad (1)$$

Each service provider must be authenticated using its certificate. The output of a service for a given call must be returned signed by the service provider, with the input call parameters. In other words, the caller must verify that the results received correspond to the data it provided.

$$P_L := C_{L-1}^1 | \ldots | C_{L-1}^n | H(C_{L-1}^1) | \ldots | H(C_{L-1}^n) \,|\, C_L \,|\, S_L^1 | \ldots | S_L^m \,|\, H(S_L^1) | \ldots | H(S_L^m) \quad (2)$$

From an input perspective, the link agent verifies the integrity and the authentication of each code used from the previous link by: first verifying the signature on the commit (for the authentication of the developer); then second, verifying the integrity of the code by comparing the received Merkle root with the one available on the blockchain. Through the blockchain, the developer can also ensure the use of the last version of the given code (freshness). Finally, third, sandboxing techniques can be used to verify and ensure all of the calls and executions that a given code makes. Moreover, the link agent must ensure the use of libraries and packages with no high-severity CVE risks.

EVALUATION AND DISCUSSION

Security requirements

An end-to-end security model for a software supply chain must fulfill numerous security requirements for the sustainability and resiliency of the ecosystem. Next, we describe the main security goals and requirements.

- *Integrity:* This involves maintaining the consistency and trustworthiness of the source code over its entire life cycle.
- *Authentication:* Only authenticated peers/developers can modify/update the source code.
- *Availability:* The source code/services must be accessible to legitimate users on demand.
- *Scalability:* The ability to ensure that the system's size has no impact on its performances. For example, if the number of users in the chain or the size of the source code used increases, the time needed for other system's functions (for example, integrity control) must not be affected.
- *Nonrepudiation:* The inability of a developer to deny having created, modified, or updated source code.

Threat model

We consider a threat model similar to the model of Dolev and Yao.[21]

Network model

The goal of an end-to-end software supply-chain security scheme is to ensure secure software development. We consider an ecosystem where numerous developers, users, code/software suppliers, and vendors (the software supply-chain actors) mainly communicate over the Internet. The network function only forwards packets and does not provide any security guarantees, such as integrity or authentication.

Attacker model

We assume that an attacker or a malicious user can modify/alter the network traffic arbitrarily with negligible delay. Nonetheless, we do not make any assumptions on the rate at which the traffic can be altered. The attacker can:

- taint/modify the source code at the output of a link
- taint/modify the source code at the input of a link
- taint/modify the source code stored
- create a new source code and store it as a legitimate source code ready for use
- spoof a developer's identity to alter/modify a source code.

Security requirements' evaluation

Our model meets several security needs and requirements. At a given link level: 1) each contributor is authenticated via his/her certificate, 2) the product of each contributor is also authenticated through the signature, 3) the integrity of each code and its components is verified via its hash, 4) relying on the blockchain ensures the freshness of the code used (the use of the latest version available), and 5) the decentralization of the blockchain ensures the availability of the data needed to control the integrity and authentication of the source codes. Finally, (6) to protect against developers' keys' compromise and theft, we propose a multifactor authentication. That is, for each commit, the developer receives a nonce from a decentralized authority. This nonce must be included in the signature of the output O_L. Equation (3) describes the output O_L if the multifactor authentication is applied and replaces (1), which only considers single-factor authentication. This nonce is also stored in the blockchain with the Merkle root of the code provided. When a developer uses this code, the link agent must verify the signature using this nonce:

$$O_L := P_L \,|\, H(P_L) \,|\, \mathrm{Sign}(P_L \,|\, H(P_L) \,|\, r). \quad (3)$$

The security of each link is vital because if attackers can tamper with the output of a step or a link before it is

provided to the next one in the chain, the end-to-end security of the supply chain will be affected. Therefore, 7) our approach ensures end-to-end security. The verification of the Merkle hash at a given link ensures the integrity of all of the codes on all of the previous supply-chain links. The same problem of tampering with the input/output between the chain's steps and links is present for the services that the software service providers offer.

> THE SECURITY OF EACH LINK IS VITAL BECAUSE IF ATTACKERS CAN TAMPER WITH THE OUTPUT OF A STEP OR A LINK BEFORE IT IS PROVIDED TO THE NEXT ONE IN THE CHAIN, THE END-TO-END SECURITY OF THE SUPPLY CHAIN WILL BE AFFECTED.

Hence, 8) in our approach we propose to link the input parameters and the output results through a signature by the service provider, which is authenticated via its certificate.

To summarize, our approach is robust and resilient against the different attacks presented in the previous section.

We are aware that there are multiple works that consider the blockchain for integrity control in the supply chain. However, in our model we rely on Merkle roots for the end-to-end integrity. The blockchain (in our model) is just a decentralized database that stores the data and ensures its freshness.

Formal validation

To verify the robustness and the safety of our protocol, we performed a formal validation using Scyther (https://people.cispa.io/cas.cremers/scyther/) a tool for the automatic verification of security protocols. In Scyther formal language, each protocol is defined through "roles." A sequence of events (for example, send, receive) defines a role. The following code shows the roles' definition of the interaction between two links in our protocol.

```
usertype SourceCode;
const sourceCode: SourceCode;
hashfunction merkleRoot;
hashfunction h;

protocol endToEndSupplyChainSec
  (link, nextLink){
role link {
macro hash = merkleRoot(sourceCode);
macro signedDataHash = h(sourceCode,
  hash);
send_1(link,nextLink, (sourceCode,
  hash,
{signedDataHash}sk(link)));
}

role nextLink {
recv_1(link,nextLink, (sourceCode,
  hash,
{signedDataHash}sk(link)));
macro signedDataHash2 = h(sourceCode,
  hash);
match(signedDataHash2,signedDataHash);
claim(nextLink,Alive);
claim(nextLink,Weakagree);
claim(nextLink,Niagree);
}
}
```

The claim event types are the goals of the formal validation. We used three authentication claim types, namely "Alive," "Weakagree," and "Niagree." The event "Match" is for the code's integrity check while it is fed into the input link.

Figure 6 shows the output of Scyther after the protocol's verification. The last two columns (status and comments) show the result of the verification process (Fail or OK), and a short description. As we can see, the validation proves that our protocol ensures the authentication of contributors and the end-to-end integrity of the code.

To summarize, through this formal validation and relying on the discussion of the previous section, we show

Scyther results : verify

Claim				Status		Commer
endToEndSupplyChainSec	nextLink	endToEndSupplyChainSec,nextLink1	Alive	Ok	Verified	No attacks.
		endToEndSupplyChainSec,nextLink2	Weakagree	Ok	Verified	No attacks.
		endToEndSupplyChainSec,nextLink3	Niagree	Ok	Verified	No attacks.

Done.

FIGURE 6. Formal validation results.

how our approach is robust and resilient against the different attacks presented in the attacker model.

Today, most software products are the result of a software supply chain. The rapid evolution of the latter has led to numerous benefits, such as higher profit, code mutualization, and the optimization of lead times. Unfortunately, its complexity makes it vulnerable to various attacks and compromises that can have different consequences on the users (consumers). We argue that, to achieve an effective security solution for the software supply chain, we need a strong understanding of its security state and features. Therefore, in this work we surveyed the different models used by the software supply chain. Then, we identified the main security issues and attacks that threaten the software supply chain. We also reviewed the different approaches used to secure the software supply chain. We found that there are multiple approaches that secure the individual steps in the software supply chain. However, very few approaches in the literature have been proposed for achieving a holistic end-to-end security for the software supply chain.

The protection of the software supply chain is becoming increasingly challenging today mainly because of the complexity of the software chain itself and the number of stakeholders participating in the software ecosystem, wherein the taint of a simple step often produces a complete subversion of the final product.[5] We believe that the key to the development of a holistic and effective, secure software supply-chain ecosystem lies in the understanding of compromises, which can affect the supply chain as a whole.[5] In this context, we proposed a security approach that satisfies the main security needs and requirements of the software supply chain. That is, it ensures not only the security of each step and link in the supply chain, but also the end-to-end integrity and authentication of the software code. Our approach is effective against malicious taints. However, as it is, it is ineffective against unintended taint and code flaws, which can be exploited by attackers to execute their attacks. Therefore, to limit the unintended taints and malicious insider threats, rigorous cross validations are required where thorough integration testing must be conducted.[22] Moreover, the development of a complete unit testing[23] adapted to the development context is needed. *Unit testing* is a type of software testing in which a small piece of code is tested to see if the code works as expected. *Integration testing* is a key level of testing to find defects where software components and the system interface interact with each other. Indeed, in software testing, a viable strategy is to look for defects and failures where they are most likely to occur. It is well known that software failures are much more likely to arise where various types of interactions occur.[24] Finally, the development of a security policy is required to limit collaborations with third-party suppliers who do not have the same level of rigor in testing and security analysis as in-house code at an organization that increases the risk of vulnerabilities. Additionally, we must always encourage collaborations with rigorous/secure third-party suppliers. C

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments, which helped us improve the content and presentation of this article.

REFERENCES

1. B. Hammi, S. Zeadally, and J. Nebhen, "Security threats, countermeasures, and challenges of digital supply chains," *ACM Comput. Surv.*, early access, 2023, doi: 10.1145/3588999.
2. J. Boyens, C. Paulsen, N. Bartol, K. Winkler, and J. Gimbi, "Key practices in cyber supply chain risk management: Observations from industry," National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep., 2021. [Online]. Available: https://doi.org/10.6028/NIST.IR.8276
3. S. Boyson, "Cyber supply chain risk management: Revolutionizing the strategic control of critical IT systems," *Technovation*, vol. 34, no. 7, pp. 342–353, Jul. 2014, doi: 10.1016/j.technovation.2014.02.001.
4. J. Viega and J. B. Michael, "Struggling with supply-chain security," *Computer*, vol. 54, no. 7, pp. 98–104, Jul. 2021, doi: 10.1109/MC.2021.3075412.
5. S. Torres-Arias, "In-toto: Practical software supply chain security," Ph.D. thesis, New York Univ. Tandon School of Eng., New York, NY, USA, 2020.
6. O. Brigid et al., "Internet security threat report (ISTR)," Symantec Corporation, Mountain View, CA, USA, Tech. Rep., 2019.
7. "2021 must-know cyber attack statistics and trends," Embroker, San Francisco, CA, USA, Tech. Rep., Apr. 2021.
8. M. Willett, "Lessons of the SolarWinds hack," *Survival*, vol. 63, no. 2, pp. 7–26, Mar. 2021, doi: 10.1080/00396338.2021.1906001.
9. B. Woods and A. Bochman, "Supply chain in the software era," Atlantic Council, Scowcroft Center for Strategy and Security, Washington, DC, USA, Tech. Rep., 2018. [Online]. Available: https://www.atlanticcouncil.org/in-depth-research-reports/issue-brief/supply-chain-in-the-software-era/

ABOUT THE AUTHORS

BADIS HAMMI is an associate professor at EPITA Engineering School, 94276 Paris, France. His research interests include cybersecurity and privacy in decentralized environments.Hammi received his doctoral degree in computer science from Troyes University of Technology, France, followed by postdoctoral research at Institut Mines Telecom ParisTech, France. Contact him at badis.hammi@epita.fr.

SHERALI ZEADALLY is a professor at the College of Communication and Information, University of Kentucky, Lexington, KY 40506 USA. His research interests include cybersecurity, privacy, Internet of Things, computer networks, and energy-efficient networking. Zeadally received his doctoral degree in computer science from the University of Buckingham, England, followed by postdoctoral research at the University of Southern California, Los Angeles, CA. He is a Fellow of the British Computer Society and the Institution of Engineering Technology, England. Contact him at szeadally@uky.edu.

10. Z. Durumeric et al., "The matter of heartbleed," in *Proc. Conf. Internet Meas. Conf.*, 2014, pp. 475–488, doi: 10.1145/2663716.2663755.
11. Y. Christian Elloh Adja, B. Hammi, A. Serhrouchni, and S. Zeadally, "A blockchain-based certificate revocation management and status verification system," *Comput. Secur.*, vol. 104, May 2021, Art. no. 102209, doi: 10.1016/j.cose.2021.102209.
12. B. Delamore and R. K. L. Ko, "A global, empirical analysis of the shellshock vulnerability in web applications," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, 2015, vol. 1, pp. 1129–1135, doi: 10.1109/Trustcom.2015.493.
13. Oxford Analytica, "SolarWinds hack will alter US cyber strategy," *Emerald Expert Briefings*, Jan. 2021, doi: 10.1108/OXAN-DB259151. [Online]. Available: https://doi.org/10.1108/OXAN-DB259151
14. "Defining insider threats," Cybersecurity and Infrastructure Security Agency, Arlington, VA, USA, Tech. Rep., Apr. 2021. [Online]. Available: https://www.cisa.gov/defining-insider-threats
15. A. P. Moore, W. E. Novak, M. L. Collins, R. F. Trzeciak, and M. C. Theis, "Effective insider threat programs: Understanding and avoiding potential pitfalls," Software Engineering Institute, Pittsburgh, PA, USA, White Paper, 2015. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=446367
16. C. Hawblitzel et al., "Ironclad apps: End-to-end security via automated full-system verification," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2014, pp. 165–181.
17. T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using delegations to protect community repositories," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, 2016, pp. 567–581.
18. T. Storch, "Toward a trusted supply chain: A risk based approach to managing software integrity," Microsoft Corporation, Albuquerque, NM, USA, 2014. [Online]. Available: https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/REVmcf
19. C. J. Alberts, A. J. Dorofee, R. Creel, R. J. Ellison, and C. Woody, "A systemic approach for assessing software supply-chain risk," in *Proc. IEEE 44th Hawaii Int. Conf. Syst. Sci.*, 2011, pp. 1–8, doi: 10.1109/HICSS.2011.36.
20. S. Simpson et al., "Software integrity controls–An assurance-based approach to minimizing risks in the software supply chain," Software Assurance Forum for Excellence in Code (SAFECode), Wakefield, MA, USA, Tech. Rep., Jun. 2010. [Online]. Available: http://safecode.org/wp-content/uploads/2018/01/SAFECode_Software_Integrity_Controls0610.pdf
21. D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. 29, no. 2, pp. 198–208, Mar. 1983, doi: 10.1109/TIT.1983.1056650.
22. J. Lee, S. Kang, and D. Lee, "Survey on software testing practices," *IET Softw.*, vol. 6, no. 3, pp. 275–282, Jun. 2012, doi: 10.1049/iet-sen.2011.0066.
23. P. C. Jorgensen, *Software Testing: A Craftsman's Approach*. Boca Raton, FL, USA: CRC Press, 2018.
24. D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: An analysis of 15 years of recall data," *Int. J. Rel., Qual. Saf. Eng.*, vol. 8, no. 4, pp. 351–371, Dec. 2001, doi: 10.1142/S021853930100058X.