Received 13 March 2020
Revised 6 October 2020
Accepted 31 December 2020

Powerful authentication regime applicable to naval OFP integrated development (PARANOID): a vision for non-circumventable code signing and traceability for embedded avionics software

Joe Garcia, Russell Shannon, Aaron Jacobson, William Mosca, Michael Burger and Roberto Maldonado
(Author affiliations can be found at the end of the article)

Abstract

Purpose – This paper aims to describe an effort to provide for a robust and secure software development paradigm intended to support DevSecOps in a naval aviation enterprise (NAE) software support activity (SSA), with said paradigm supporting strong traceability and provability concerning the SSA's output product, known as an operational flight program (OFP). Through a secure development environment (SDE), each critical software development function performed on said OFP during its development has a corresponding record represented on a blockchain.

Design/methodology/approach – An SDE is implemented as a virtual machine or container incorporating software development tools that are modified to support blockchain transactions. Each critical software development function, *e.g.* editing, compiling, linking, generates a blockchain transaction message with associated information embedded in the output of a said function that, together, can be used to prove integrity and support traceability. An attestation process is used to provide proof that the toolchain containing SDE is not subject to unauthorized modification at the time said critical function is performed.

Findings – Blockchain methods are shown to be a viable approach for supporting exhaustive traceability and strong provability of development system integrity for mission-critical software produced by an NAE SSA for NAE embedded systems software.

Practical implications – A blockchain-based authentication approach that could be implemented at the OFP point-of-load would provide for fine-grain authentication of all OFP software components, with each component or module having its own proof-of-integrity (including the integrity of the used development tools) over its entire development history.

Originality/value – Many SSAs have established control procedures for development such as check-out/check-in. This does not prove the SSA output software is secure. For one thing, a build system does not necessarily enforce procedures in a way that is determinable from the output. Furthermore, the SSA toolchain itself could be attacked. The approach described in this paper enforces security policy and embeds information into the output of every development function that can be cross-referenced to blockchain

The authors wish to thank the Naval Air Warfare Center Aircraft Division's Naval Innovative Science and Engineering (NISE) program for their support of this work.

Journal of Defense Analytics and Logistics
Vol. 5 No. 1, 2021
pp. 46-76
Emerald Publishing Limited
2399-6439
DOI 10.1108/JDAL-03-2020-0006

transaction records for provability and traceability that only trusted tools, free from unauthorized modifications, are used in software development. A key original concept of this approach is that it treats assigned developer time as a transferable digital currency.

Keywords Software development, Blockchain, Cybersecurity, Operational flight program, Secure development environment, Secure virtual machine, Zero trust, Embedded systems, Mission-critical systems, OFP, DevOps, DevSecOps, Software support activity, SSA, SDE, Permissioned blockchain, Cryptocurrency, Time-limited authorization for developer action, TADA, Code signing, Trusted software guard, SGX, Trusted eXecution technology, TXT, Trusted platform module, Self-hosting, Controlled access blockchain, CABlock, Role-based access control, RBAC

Paper type Conceptual paper

1. Introduction

Software has become the most significant component of a modern combat aircraft's capability. This is a trend consistent with many types of weapon systems (Brown and Hurt, 2017). A cyberattack on US Naval avionics software systems and on the avionics software supply chain is disturbingly feasible. Senior Navy leadership has stated that "Adversaries can look for vulnerabilities in the software, supporting systems or supply chains to disrupt and sabotage operations" (Yasin, 2018). In Naval Aviation, the focus of the adversary would be the avionics program load, known as the Operational Flight Program (OFP), along with its software distribution network. Suggested mitigations of this immediate threat to the software supply chain include hashing (National Institute of Standards and Technology, 1994, 2015), code signing and the use of Hash-based Message Authentication Code (HMAC) (National Institute of Standards and Technology, 2008).

The authors posit that securing the supply chain will ultimately drive the adversary to attack the software development organization, henceforth referred to as the Software Support Activity (SSA). This paper presents a model for embedding long-term security in the code development process of an SSA.

Software Development Operations or DevOps is the implementation of development under an Information Technology (IT) system. DevOps is the business of the SSA. DevOps with integrated security is known as "DevSecOps." Recently, there have been efforts to bring about effective capabilities in DevSecOps for DoD, most notably the effort of the US Air Force called Platform One, under the Department of Air Force Chief Software Officer. This project has provided a number of open-source DevSecOps tools (Platform One, 2020).

It is the intention of PARANOID to complement and enhance above such efforts, not just by more and better adherence to the conventional cybersecurity best practices such as the Risk Management Framework (National Institute of Standards and Technology, 2017) and associated controls (National Institute of Standards and Technology, 2013), but rather to provide the ability to prove at the point-of-load, on a module-wise basis, that said the software was developed on tools not subject to unauthorized modifications and that it was developed by approved developers on approved development hosts on the approved schedule. This capability involves the utilization of blockchain, making each development action a transaction such that module-wise authentication is based on vetting blockchain transaction records. PARANOID, however, is more than simply applying blockchain technology at the SSA. Rather, it is the judicious combination of a blockchain with a secure and trusted development environment that enforces transactions based on development. Thus, PARANOID becomes the basis for proving the integrity of the development system at the point of software load by

comparing module signatures inserted by the environment throughout the course of development that can be cross-referenced with blockchain transaction records. The intent is to push this approach to a "paranoid" degree, such that from the binary, it is possible to trace back to authenticate the source code and the source code history overall edits, developers, the developer management chain, any development tools for embedded code and furthermore, the tools used to develop the tools. All of this is achievable from embedded signing and blockchain. This solution can coexist with other fielded security solutions or those under development, adhering to the defense-in-depth best practices (United States Department of Defense, 2019).

1.1 The threat to the software support activity

In 2017, Chinese hackers attacked the development system for CCleaner such that subsequent builds contained malicious functionality which was distributed to thousands of customers via automatic update (Saarinen, 2017; Shah, 2017; Higgins, 2018). A variant of this attack was used against the NetSarang database product development system (GReaT, 2017). Additionally, there are multiple documented methods of how malicious actors can introduce malicious code at the SSA, resulting in the compromise of previously-trusted code (Constantin, 2017; Burton, 2017; GReaT, 2017; Gegick and Barnum, 2005; Anderson *et al.*, 2004; CBS, 2015; Wheeler, 2017).

2. The PARANOID method

For digital currencies to be viable, they must support the means to detect fraud, *e.g.* counterfeiting and prevention of double-spending (World Crypto Index, 2018). The authors posit that detecting unauthorized code modifications is, at some underlying level, similar to the problem of detecting and preventing fraud in digital currency systems. Furthermore, it will be shown that distributing a currency to various parties in an organization is a way to distribute authorization. For the SSA, this would be authorization to perform development. The ability to capture digital currency transactions on a blockchain provides for a non-reputable record of digital currency transactions. This will be the basis of provability and traceability in PARANOID's security model. The Department of Defense (DoD) recently included the use of blockchain technology in its 2019 Digital Modernization Strategy to meet network communications security goals (United States Department of Defense, 2019).

2.1 The software support activity security model based on the TECHS condition

The security model of the SSA, in part, relates to who authorizes coding, who is permitted to develop code, where the code will be developed and in what period of time the code development will take place. Secure development consists of what is called the "TECHS" condition:

- Tools and Equipment: Use of approved (known good) development tools or toolchains, (*e.g.* editors, compilers, *etc.*) on the approved development equipment (*e.g.* host, server, network, *etc.*).
- CHain of authority: In corporate and government organizations there is a management hierarchy, whereby a supervisor authorizes a developer to do programming work on a project. The supervisor reports to a superior, someone mid-level in the organization. This reporting relationship possibly repeats some number of levels, eventually leading to a High-Level Authority (HLA) from which all authorization to perform development originates.
- Schedule: The code was created/developed/modified/obtained during the officially-approved development schedule.

In an ideal world, the TECHS condition holds true. However, in the real world of ever-present cybersecurity attacks, an attacker violates one or more of these conditions. For instance, if the attacker makes a malicious modification to source code or to a binary after a development schedule has officially ended, the attacker is operating outside of the approved schedule. By implementing malicious functionality that is not intended by management, the attacker is operating outside of the chain of authority. The attacker is likely to use unauthorized tools (toolchain) and equipment (development hosts/development network). What the cybersecurity community needs is a system that enforces the TECHS condition. PARANOID enforces the TECHS conditions with three primary components: a secure development environment, a blockchain and an authentication mechanism.

2.2 The foundation: implementing the secure development environment

2.2.1 Secure development environment combined with blockchain-represented transactions as the basis of security. A secure development capability on the development system is the necessary foundation for trusting the integrity of development transactions represented on the blockchain. Specifically, the authors want a development system that makes both the embedding of signatures, and the subsequent dispatch of the transaction information to the blockchain processing nodes, mandatory operations associated with every critical development function.

To achieve this, the authors propose a Secure Development Environment (SDE). The SDE, as envisioned, would be implemented as a secure Virtual Machine (VM) or secure container, operating on a host with various security capabilities configured to protect the VM with regard to integrity. These capabilities would include a Trusted Platform Module (TPM) (TCG, 2020) for systems using Intel Corporation's x86 central processing units (CPUs), Intel Trusted eXecution Technology (TXT) (Intel Corporation, 2019a) and Software Guard eXtensions (SGX) (Intel Corporation, 2014, 2016b). For systems using ARM, Trust Zone (Coombs, 2015) would be used for implementing a Trusted Execution Environment (GlobalPlatform, 2008). All the aforementioned technologies are capabilities that can provide an attestation of application integrity, *i.e.* proof that the loaded memory image of the application, in this case, a development tool, has not been subject to unauthorized modification even if the host is compromised at the kernel privilege or higher. TXT uses authentication keying material from the TPM chips, which is securely transferred to the processor, while SGX uses keying material derived from a master key stored directly on the processor for attestation.

The SDE is a *production* virtual machine or container, with an associated hypervisor/container management package that would ideally come pre-built from a trusted source and would be installed on the target development host without any modification. It would come preconfigured with the development toolchain. As illustrated in Figure 1, the SDE has two major storage volumes. One volume contains the toolchain and the operating system (OS). This volume is read-only and is not accessible either from the host or from the SDE itself. The other volume contains the development code. It is writable from the SDE and read-only with respect to the development host.

2.2.2 A self-hosting secure development environment capability. The ultimate vision for the PARANOID concept is as a trusted source of DevSecOps toolchains, incorporated into SDEs that themselves are represented on a blockchain.

Being blockchain-enabled and containing a repository of its own source code for the toolchain and OS components, in this arrangement, the SDE will have the ability to recompile itself to create an identical SDE with hashes that could then be cross-referenced with a blockchain for proving that the built version is identical to the SDE used in implementing the

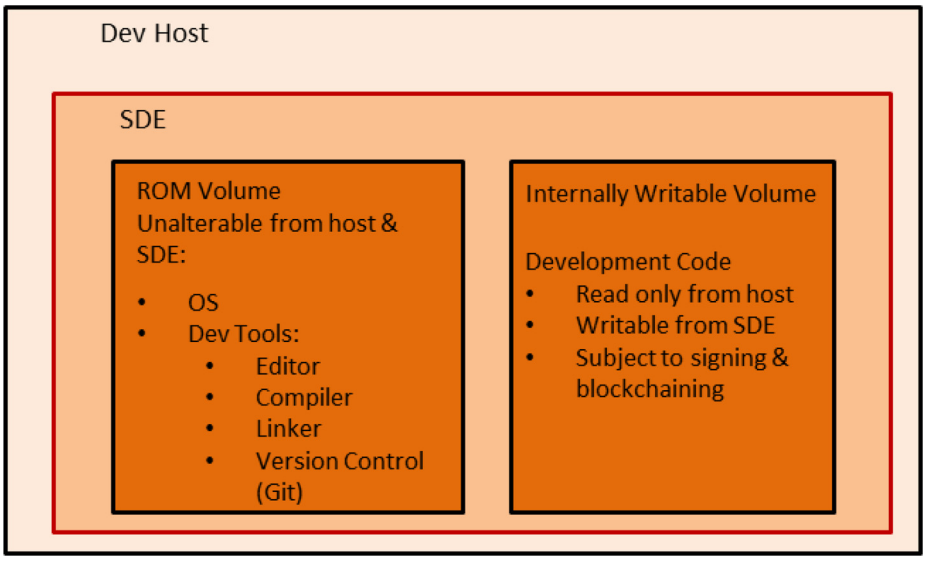

Figure 1.
The secure development environment (SDE) inside a development (Dev) host

build. This capability enables the toolchain to further develop itself, thus providing a self-hosting capability. In this way, the development of the SDE itself would be a PARANOID process, *i.e.* the SDE is the development system for its own development and with the blockchain, supports an indelible record of its development. The SDE as a self-hosting property should be achievable on open-source Linux-based VMs or containers, in conjunction with an attestation capability from the host CPU. This SDE self-hosting property is a significant strength of the PARANOID approach with regard to providing strong traceability and provability of the SDE/toolchains with regard to the TECHS condition. Considering the aforementioned, with regard to the classic Ken Thompson attack on a compiler (Thompson, 1984), PARANOID should provide strong provability and traceability against such a threat.

Should PARANOID gain sufficient traction as a viable DevSecOps capability, it may be appropriate to eventually have an SDE developer/maintainer agent working with the Navy (or even at the DoD level, should other services become interested) to manage the development of the SDE and support the vetting of the toolchain, blockchain and OS components for incorporation into the SDE. Furthermore, this entity would continuously improve the SDE with its own developer staff, using the SDE as the development system for itself in the above described self-hosted manner, with all subsequent modifications to the SDE recorded on the blockchain. This entity would periodically release new SDE versions for distribution to various avionics SSAs where these releases would be subsequently used for embedded software development. Thus, the self-hosted SDE developed under the PARANOID process will provide for known secure SDEs with the toolchain itself having full traceability.

2.2.3 The secure development environment provides signatures for weapons replaceable assembly authentication and other functions. Within US naval aviation, major avionics units containing electronics are referred to as Weapons Replaceable Assemblies (WRAs). In WRAs, there are custom electronics boards residing on a backplane that perform low-level functions related to specific avionics systems such as transmitters, receivers, sensors, navigation and control systems that are implemented with digital and analog circuits. WRAs

may also contain one or more Single Board Computers (SBCs) whose coordinated workings with the custom boards provide the avionics functionality. An OFP is commonly a bundled group of binary images in which each image individually targets a specific SBC. Under PARANOID, the binary signed modules are compiled from their corresponding source code modules. The signature might reside in an unused section of the binary header. For example, consider the Linux Executable and Linkable Format (ELF) binary structure, in which a signature block might be implemented in part of the .note section (Linux Foundation, Tool Interface Standard Committee, 1995). The specific SBC target binary image may have components such as Real-Time Operating System (RTOS) software, user-mode control code or User Generated Software (UGS), Board Support Package (BSP) code consisting of board-level drivers or other software which, in combination, implement the specific avionics functionality for that particular SBC. As shown in Figure 2, using PARANOID, each of the individual modules will include a signature block in addition to the standard binary segments (*e.g.* text, data, .bss, stack and heap). The expanded signature includes the build signature of the editor, a second signature from the compiler and a signature added during the build (linking) process. The initial signature is inserted by the editor upon each edit (or commit) into a special demarcated section of the source code file. In PARANOID, every development function involves the exchange of Time-limited Authorized Developer Action (TADA) represented in blockchain records. Assigned development time in TADA is what "pays" each transaction. More will be said about this in a later section. Upon compilation, the edit signature embedded in the source code is accessed by the SDE compiler and automatically inserted into the signature block of the complier's output object file. The hash of the object file is signed by the SDE compiler signing function along with the signature (signed hash) obtained from the source code. The same thing occurs at the linking stage concerning the executable binary. It is these signatures, along with their corresponding hashes, that are used to check against blockchain records of development transactions.

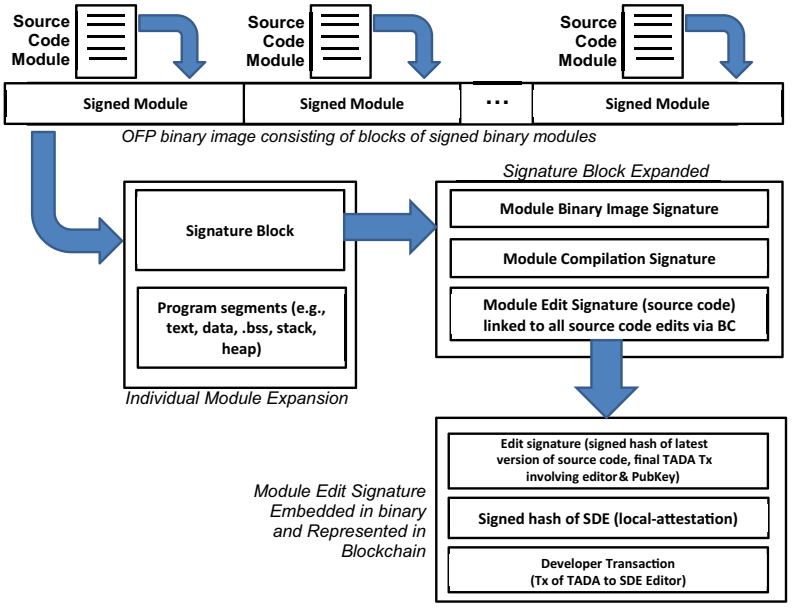

Figure 2. Authentication signature including the transaction of TADA that "paid" for the edit

2.3 Approach to incorporate software from external sources into PARANOID-based DevSecOps

At the beginning of avionics software development and during subsequent software upgrades, original SSA-developed software code or software code from third parties, *e.g.* open-source code or procured commercial-off-the-shelf (COTS) code, may be incorporated into the avionics code base. Anticipating that PARANOID will need to authenticate any newly introduced software, the HLA can arrange for some number of signed, "dummy" source code template files containing no actual functioning source code. These template files can be represented on the blockchain as place-holders for future modifications, *i.e.* adding functional source code. Each template would contain only the embedded signatures and a randomly-generated Universally Unique Identifier (UUID). To bolster security further, individual developers could be restricted to certain lineages of source code based on assigned UUID series such that template files are blockchain represented, possibly corresponding to different lines of embedded software development, *e.g.* one line for navigation software, a different line for fire control software, *etc*. This restriction might work with corresponding lines of TADA for each of the aforementioned lines of development with associated template files.

For the introduction of third-party pre-existing source code into the code base during an upgrade or for a new start avionics software project, the SSA would need to verify the integrity of the outside source code before incorporation. Then, an SSA-enforced code entry or enrollment procedure will be required to introduce the code to the above-mentioned templates using a PARANOID SDE editor. Making peer review a critical development transaction via a ticketing process would be a means to vet outside source code being introduced into the SSA code base and would be a means to defend against dangerous code being "copied-and-pasted" into the SDE editor. Alternately, or in addition to the above, making the application of static analysis tools or automated vulnerability detection tools a critical development function for the above code incorporation would be another approach to support outside software code entry.

2.4 Use of a digital currency key to implementing PARANOID

To enforce TECHS, PARANOID will use attributes of cryptocurrency in making critical stages of software development a transaction (Tx), where cryptocurrency is exchanged. Critical stages subject to transactions may include:

(1) Developer Assignment
(2) Editing
(3) Compiling
(4) System builds, *e.g.* linking and generating the OFP or program load format binary and possibly other stages of development such as:

- Generation of a board-level storage format binary image.
- Creation of a packed binary, implemented during development with a compression utility applied to the main board-level binary format.
- Creation of MIL-STD-2217 (United States Department of Defense, 1991) compliant loader image format (OFP load format), consolidating individual binaries targeting multiple boards into a single Rapid Reprogramming Terminal file format (.rrt). This involves additional packing in conjunction with

combining of board-level and/or subsystem packed binaries having additional headers inserted to describe the delivery of specific binary packed images to target SBC boards in WRAs, also adding CRC fields for corruption detection.

Other stages of development that optionally could be implemented as a transaction:

(1) Automated source code analysis (static analysis), using tools such as Polyspace (The MathWorks, Inc, 2017) or CodeSonar (GrammaTech, Inc, 2018).

(2) Automated vulnerability discovery software, *e.g.* identification of Common Weakness Enumeration (CWE) (The MITRE Corporation, 2019).

(3) Compliance testing software, *e.g.* a compliance test suite for:
   - Future Avionics Compatibility Environment (FACE) (The Open Group, 2016).
   - Hardware Open Systems Technologies (HOST) (Collier, 2016).

(4) Ticketing, issue resolution tracking, *e.g.* using Jira (Atlassian, 2018).

(5) Software source code peer review (may be implemented by ticketing).

(6) Formal verification (must develop in a language subject to mathematical proofs, *e.g.* Haskell [Haskell, 2020]), here an "abstract specification" relatable to source code has properties of confidentiality, integrity and availability mathematically proven using an assisted proving tool (Klein *et al.*, 2009, 2014).

With the above, we have:

- Development host CPU-based attestation (Intel Corporation, 2016b) of all toolchain software involved in the development transactions, *i.e.* each development host implements signing such that not only the individual developer is represented in the signing process but also the specific development host used by the developer, where the host CPU "attests" to the integrity of the tool software as loaded in system memory. Thus, the transaction message dispatched to the blockchain processing nodes accurately represents that a critical development function was performed by a tool not subject to unauthorized modification on a known approved development host.
- Transaction history of all development transactions is accessible from a system-wide accessible blockchain, created by a network of blockchain processing nodes that might be implemented as a system of interacting virtual machines or containers, on an enterprise host network.
- Endpoint loading with the ability to test OFP per module for TECHS compliance based on a blockchain Merkle proof (blockchain-based authentication). Merkel proofs will be discussed further in Section 3, below.

2.5 Digital currency, transfer of authorization, alloted developer time and the schedule enformcement

In the PARANOID method, there is a quantity of digital time-limited authorization, represented on a blockchain, with specific attributes to convey authorization and enforce the schedule ("CH" and "S" of the TECHS condition). Because of the transactional nature of the transfer process, the method has a currency-like quality. The authors have named this quantity Time Limited Developer Action (TADA). TADA could be implemented in units roughly equivalent to time units representing assigned development time. For example, a unit of TADA could represent some unit of assigned development time. Every development

action will then result in the transfer or expenditure of TADA. The following properties are associated with TADA:

(1) Hierarchical distribution featuring:
   - Creation at top level for multiple software engineering projects by an agency or organization High Level Authority (HLA) such as the Under Secretary of Defense for Research and Engineering (USD(R&E)) or service equivalent, *e.g.* the Assistant Secretary of the Navy for Research, Development and Acquisition (ASN(RDA)) or the Department of the Navy Chief Information Officer (DoN CIO).
   - Subsequent to creation by HLA, distribution from HLA to mid-level authorities (MLAs) of an agency, or organization, *e.g.* a Chief Engineer (CHENG) or Chief Information Officer (CIO), for Navy systems commands (SYSCOMs) or Navy program offices or a Navy Warfare Center mid-level official with cognizance over software development for a specific program office.
   - Continuing down the hierarchy, distribution to software development supervisors within a specific project.
   - At the lowest level of the hierarchy, distribution to individual developers on a specific project.

(2) Represents time allotted for software development. As there is an expiration date, TADA can only be used for a limited time. This time limit could be implemented via hard logic in the blockchain processing node or possibly by smart contracts [1] associated with the specific TADA transactions in a general-purpose blockchain that supports smart contracts. The "type" of TADA could also limit the specific area of software development, as will be discussed in Section 4.2.

(3) Allocated through an organization's management chain via distribution transactions.

(4) Every stage of software development is a transaction involving the transfer of TADA.

(5) Every developer action, *e.g.* editing, compiling, *etc.*, results in some amount of TADA expenditure, *i.e.* development time is being charged.

(6) TADA transactions as represented on the system-accessible blockchain:
   - Provide secure verification that code development was performed on authorized hardware and development tools based on attestation (TECHS item 1).
   - Represent a securely-verifiable chain of authorization (TECHS item 2) all the way from HLA to binary.
   - Provide secure verification that only code modifications were performed during the authorized development period (TECHS item 3).

(7) A key attribute of the cryptocurrency basis for PARANOID is Byzantine Fault Tolerance (BFT) (Lamport *et al.*, 1982), whereby any attempt to double-spend TADA, *i.e.* an attempt to produce two different development transactions using the same TADA, will be detected and halted.

2.6 Implementing time-limited authorized developer action

Implementing TADA with real cryptocurrency, where TADA is mapped to nominal amounts of real cryptocurrency could support some uses of smart contracts. However, it might be more practical and more in line with organizational security practices to implement

a non-public enterprise-based system of exchange consistent with a private permissioned blockchain system (Bryson *et al.*, 2017).

TADA will be created on a server controlled by the HLA and then distributed via Mid-Level Authorities (MLAs) across an organizational chain of authority, as illustrated in Figure 3. Each distribution successively breaks the allotment of TADA into smaller chunks until it reaches first-line supervisors (Sups), where it is ultimately assigned to the developers (Devs). The units shown in the figure are notional, representing one man-year (in seconds) at the developer level.

Here is a specific example using a series of typical TADA transactions in a development project, beginning with the HLA and ending with the developer. Figure 4, below, is in the style of Satoshi Nakamoto's original 2008 whitepaper (Nakamoto, 2008), in which the concept of a blockchain was first described. The figure illustrates the HLA generating a TADA Creation Message (TCM) on a TADA Creation Server (TCS). If implemented as a private digital currency, this TCM might set a deadline for the expenditure of TADA, thus defining the period of authorization for coding. If public cryptocurrency is used, the TCM may incorporate the use of a smart contract to set the authorized period of performance. Another alternate might be a hybrid of private and public cryptocurrency, whereby nominal amounts of public cryptocurrency are purchased by the HLA and treated like a number of property deeds on a public blockchain to be bought and sold via cryptocurrency (Szabo, 1998). Whether public or private blockchain is used, a "wallet" would securely store transaction credentials (Rosic, 2018). The wallet of credentials could be implemented on the HLA's local secure host or externally on a smart card (Secure Technology Alliance, 2018)

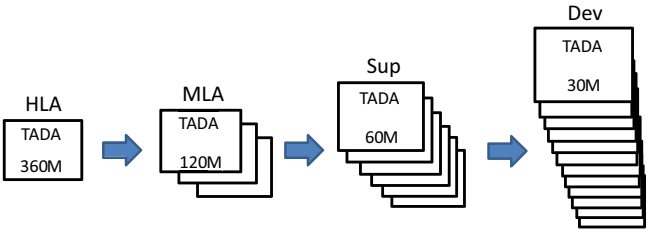

Figure 3.
Distribution of 360 M units of time-limited authorized developer action (TADA) from a high-level authority (HLA) through mid-level authorities (MLAs) and supervisors (Sups) to the developers (Devs)

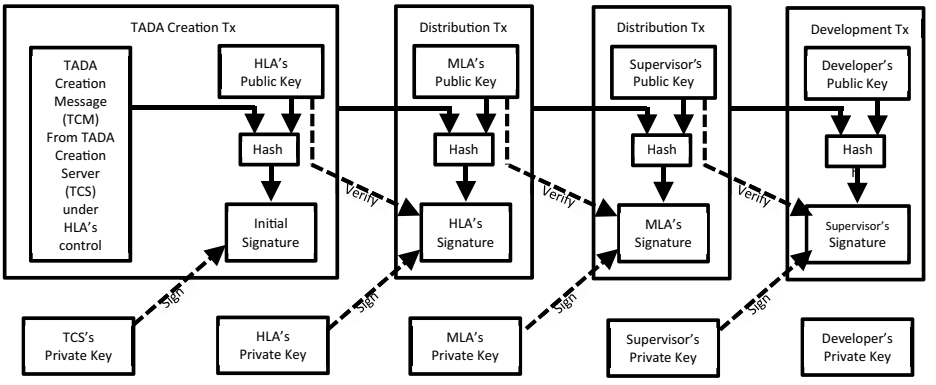

Figure 4.
TADA distribution from HLA to developer

such as the DoD's Common Access Card (CAC) (Defense Human Resource Activity, 2018). Like all cryptocurrencies, the TADA itself would not be stored in the wallet. Rather, any party's balance is determined from the original allocation and tracked according to the sum of transactions contained on the blockchain.

An actual TADA distributional transaction, in the form of the blockchain dispatch message from the SDE, might look like the following prototype code.

```
[
 [
  'Key',
  'hello_world.c'
 ],
 [
  'Record',
  {
   codeHash: 'b8639f4970c456e4502827d4af7ee7ee4ec25ca4b8809268649f
   5580248d73a0',
   lastHash: '1e4857ca70dd716be8ee876e821a49001adc4f251ad664c12b95311e
   9411dbb5',
   tadaTokenToAddress: 'someTADAToAddress',
   tadaTokenFromAddress: 'someTADAFromAddress',
   tadaTokenAmount: 897,
   devPublicKey: 'demo_user',
   sdeVersion: '1.0',
   revision: 'aa6198d2e4ca0187740766f54262de61a376764b (Wed Aug 5 18:00:22
 2020 + 0000)',
   enclaveHash: '4c3774ab5bc2e45b59c0ccc2 f91f686b9e467405509671
 fbad82709d0623cfc3'
  }
 ]
]
```

Information pertaining to a certain development action will be uniquely identified with the file's fully-qualified name as shown in the "Key" data field. The "codeHash" and "lastHash" fields represent the file's current SHA-256 hash and the SHA-256 hash of the previous entry for that file, respectively. The "tadaTokenToAddress" field corresponds to the TADA address that will receive the TADA charged to the developer for this edit transaction. The "tadaTokenFromAddress" field corresponds to the developer's TADA address and is the TADA address charged for the transaction. The "tadaTokenAmount" field is the amount of TADA the developer is charged for the transaction. The "devPublicKey" field is the developer's public key that is used to connect to the PARANOID network. The "sdeVersion" field contains the current version of the SDE. Finally, the "revision" and "enclaveHash" fields show the attestation information.

Each TADA transfer transaction would result in a message like the one shown above, which would be sent to the various transaction processing nodes on the network supporting the blockchain. The transaction would be entered into a blockchain block with a distributed consensus algorithm that will be used to select the next valid and universally recognized, block on the blockchain. There are a variety of distributed consensus algorithms available (Dwork and Nair, 1992; Jakobsson and Juels, 1999; Bitcoin Wiki, 2018b, 2018c; King and Nadal, 2012; Digiconomist, 2018; Vasin, 2014; Buterin, 2014b, 2014c, 2014d).

Figure 5 illustrates two blocks on a blockchain in the style of Nakamoto's paper (Nakamoto, 2008). The blockchain would contain unalterable transactional data for all development functions. Over time, it may consist of a large number of blocks. If a public blockchain is used, this would be shared with all sorts of commercial transactions and would be many tens of gigabytes or more worth of data. A more secure solution would be a private, permissioned blockchain restricted to enterprise transactions or a blockchain used exclusively for development by the Navy or by the Department of Defense. An intermediate permissioned system might be an enterprise blockchain that supports multiple functions dedicated to a specific enterprise supporting multiple functions, including development.

Once a blockchain is created, it is continuously added to and accessed by a group of processing nodes known as the "blockchain network." At its core, a blockchain network provides a group of users with security services that are defined in terms of reading transactions and updating transactions to a secure data ledger that is the blockchain. For our previously discussed application of distributing developer time like currency, the authors would like to implement secure transfers of developer time, such that one of the interfaces defined for this network would be an interface that permits the transfer of developer time. "Chaincode" is a specialized program that securely implements an interface on the blockchain. In the example above, the chain code would execute defined logic and securely write to the ledger that would subtract the TADA payment from a TADA distributer and then add it to TADA recipients. This network consists of nodes, each running in its own secure, isolated containerized environment and is the sole mechanism to update the underlying data ledger tracking the state of the network.

Consider the developer performing an edit, as illustrated in Figure 6. When editing source code through the development host's SDE, some amount of TADA is expended with each edit and with the creation of a transaction receipt which builds another link onto the blockchain, the SDE itself becomes a party to the transaction. Similar transactions occur with other critical development functions such as compilation, linking, conversion to OFP load format and compliance testing. The final transaction of TADA involving the transfer of a charged amount of TADA to the SDE is arranged at the time of saving an edit (or making a commit). This final transaction occurs inside the SDE and is distinct from the previous distributional transactions due to its multi-stage nature. Proceeding from left to right in Figure 6, below:

- The most recent edit was performed on the SDE (proven by SDE signature and corresponding blockchain transaction record) (TECHS item 1).
- The SDE is free of alterations (proven by local [self] attestation signature and corresponding blockchain transaction record) (TECHS item 1).
- The line of distribution transactions of TADA from the HLA to the developer and subsequent expenditure on the SDE by performing the edit proves that the developer who is authorized by the organizational management chain wrote the

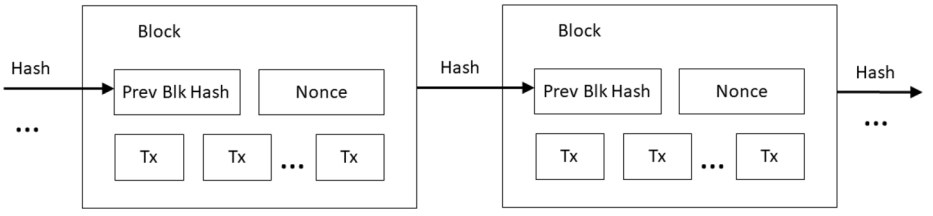

Figure 5. Blockchain representing TADA transactions (Tx) supporting SSA development functions

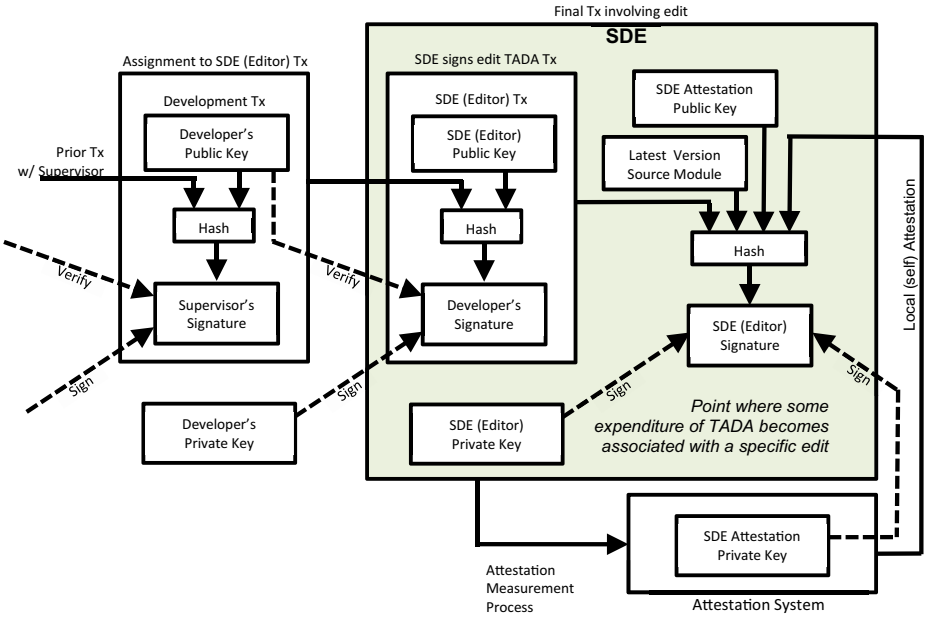

Figure 6.
Expenditure of TADA from developer to SDE relating to an edit

code within the authorized schedule (proven by all signatures and corresponding blockchain records, *i.e.* all applicable transactions occurred prior to TADA expiration and thus were accepted into the blockchain) (TECHS items 2 and 3).

One might ask, "How much TADA does a specific development action cost?" In other words, how is TADA charged? The intent of PARANOID is not to represent the value of development functions but rather, generate transaction records for them. One approach for a cost model consistent with the aforementioned is the Charge Above Remaining Time (CART) model. This would work as follows:

(1) Every developer action costs TADA units above remaining time in the assigned development period of performance.

(2) Assume TADA units correspond to the time in seconds.

(3) For example, assign a developer to have a particular task completed in one year:

- One calendar year of TADA equals 31,536,000 seconds assigned to the developer (via the chain of authority). Because of the nature of CART, one could assign a calendar year's worth of time. It would not be based on hours actually worked, but rather a time to the deadline.
- If a developer waits six months before performing his or her first action, *e.g.* an edit (type in a few lines of code and save), that action is charged 15768000 TADA, *i.e.* one half of a year of seconds.
- If the same developer then completes another edit in 60 seconds, the subsequent action is charged 60 TADA.
- Every subsequent action is charged above the time left before the TADA expiration.

(4) No charges are allowed after the deadline has been reached.

(5) If development work proceeds to the end of the authorized development period, this charging model guarantees that the TADA allotment will be zeroed out. If the development takes longer than anticipated, more TADA can be allocated. Remember that TADA does not represent value. It is a basis for transactions that enforce deadlines.

(6) If the development completes early, the supervisor can recall TADA, which would freeze code development.

The example above is, of course, a simplification of how source files are created and modified in a major software development program. In modern development systems, a versioning system such as Git (Git webpage, 2019) is used and usually integrated with an editor. The actual process would be implemented at the "commit" stage of the versioning tool, whereby the source is incorporated into a Git repository or "repo" and a signed hash representing the source code file is generated. Though using hashing and signatures implies security, in this case, they are intended for corruption detection and not security. Other developers such as Ellcrys (Idialu, 2019), are working on a distributed Git repo technology that uses blockchain technology to implement secure connectivity across nodes that comprise the distributed repo.

It would not be fair to omit a mention of the many development systems with some degree of tracking and traceability for source code modification. Existing systems have "check-out" processes whereby a developer gets permission to make modifications to source code. Upon completion, the modified source code undergoes "check-in" and can be used for a build. What distinguishes PARANOID from these systems is that a strict security policy is integrated into the complete software development work-flow. Existing development systems may track changes in source code, but the build system may still allow a build with improperly checked-out code modifications. It is intended that PARANOID will automatically and completely enforce the security of the development system, such that code with unauthorized modifications will not be allowed to be built.

3. Using PARANOID for endpoint operational flight program authentication

"In my opinion, it's all about the Merkle Tree."

– Vitalik Buterin, chief scientist of Ethereum, DEVCON1 (Buterin, 2016).

As described previously, a distributed consensus algorithm is the basis for determining how transactions are arranged into a block and how the next blockchain block is identified. A block is identified by its hash. In Bitcoin and other blockchains, if the root hash of a block appears in the subsequent block, that is compelling evidence that the previous block is valid, so the new block is accepted into the blockchain (Nakamoto, 2008).

3.1 Recognizing an authentic operational flight program load at the endpoint via a binary Merkle tree proof

Consider a block on a conceptual PARANOID blockchain, as illustrated in Figure 7, representing the *ith* block and illustrated in the style of Nakamoto (2008). In this diagram, there are only four transactions in the block. The number would be considerably more in a real blockchain block such as the Bitcoin blockchain (The number of transactions per block is variable in Bitcoin, but on average, it is between 1,500 and 2,500 transactions per block, depending on the overall block size [Blockchain Luxembourg S.A., 2020a].) At the bottom of Figure 7 are the transactions representing various development actions in the order they come into the blockchain network. Immediately above that are the corresponding hashes.

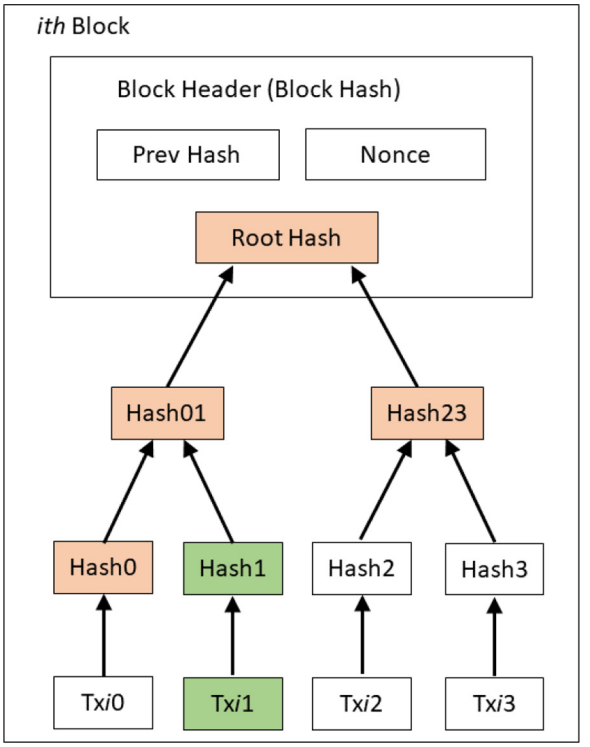

Figure 7.
Binary Merkle tree (BMT) on a blockchain block

Each hash is paired with a nonce (an arbitrary, often pseudorandom number issued in an authentication protocol to ensure that old communications cannot be reused in replay attacks) and the pairs become arguments for subsequent hashes proceeding up, in a hierarchal manner, until a single, top-level hash is achieved for the block. This top-level hash is the root hash of the block. A successive pairwise hierarchical hashing like this one is called a Binary Merkle Tree (BMT), where the transactions are the leaves and the top hash is the root hash.

The pairwise arrangement of the block emerges from the binary-decimation nature of certain hash algorithms, where the output size is half the size of the input. For example, for SHA-256, each input is 512 bits (64 bytes) and each output is 256 bits (32 bytes). Two 256-bit outputs of two hashing operations on the previous level are combined, providing a 512-bit input to the next level and so forth. It is necessary to calculate the BMT to generate a root hash of a block. The number of operations scales at order $n\log_2 n$ hashing operations for n transactions, assuming a transaction hash is supported by standard hash functions such as SHA-2 (National Institute of Standards and Technology, 2015). Note also that obtaining the hash for each transaction would take several rounds of hashing (of hashing hashes) because an individual transaction in a tentative block is considerably larger than 512 bits (64 bytes). For a Bitcoin-sized transaction, the transaction message is around 256 bytes (Stack Exchange Inc, 2014; Bitcoin Wiki, 2018d), about four times the SHA-256 input size, multiple hashing rounds $4 \times log_2(4) - 1 = 7$ SHA-256 hash operations would be needed to effectively calculate an

individual transaction hash. PARANOID may require a larger transaction size compared to Bitcoin, especially for the critical development transactions because of all the signature fields that must be represented in the transaction. Once the individual transaction hashes are calculated, one can then proceed to implement the Merkle Tree such that the block hash can be subsequently be calculated. For a Bitcoin-sized block of around 2,000 blocks (Blockchain Luxembourg S.A., 2020b), this would be approximately $2048 \times log_2(2048) - 1 = 2048 \times 11 - 1 = 22{,}527$ hash operations.

To verify Tx*i*1 in Figure 7 and to complete the proof, all that is needed at the confirming endpoint are the items in Figure 7 in orange (with Hash0 and Hash23 with Tx*i*1, the root hash of the block can be calculated, which is of course listed on the block). This is known as the Pruned Binary Merkle Tree (PBMT) representation. Working out all the hashes along a PBMT is called a Merkle proof (Nakamoto, 2008).

The transaction records Tx*i*0 through Tx*i*3 are the low-level leaves in a hierarchy of successive hashing levels, culminating in a root hash. The magenta-colored elements in Figure 7 represent the Pruned BMT (PBMT) for verifying the Tx*i*1 (orange) transaction with the root hash. Besides Tx*i*1, the verification requester would only have to provide hashes Hash0 and Hash23 to the receiving LWPN such that it could confirm that Tx*i*1 is part of the *i*th block. The LWPN has to calculate the block root hash which provides the verification that Tx*i*1 is a valid transaction based on the root hash being accepted into the blockchain by the existance of successive blocks. This is a powerful capability to provide for proving the validity of any developmental transaction.

The hash trees can extend above the blockchain blocks. There could be "blocks of blocks" that themselves are hashed. This would not be of interest in a cryptocurrency system but is relevant to software security in authenticating a software upgrade whose development is represented over many blockchain blocks. The goal is to prove, via Merkle proof, that for some binary module in the upgraded software, the binary having been compiled from its associated upgrade source code module (as in Figure 2), that the upgraded source code itself is derived from a corresponding earlier version of source code previously compiled to the binary and currently resident on the endpoint.

Note that the term "binary module," which we have used to refer to a section of the binary image (build output) compiled from a specific source code module (function, file), could just as easily refer to the module code in the form of the packed binary format for non-volatile storage (flash) memory or the specific OFP load format for loading into its target endpoint. In whatever form, the module is compiled from a corresponding source code module. Remember, too, that with the embedded signatures, the corresponding source code module is directly traceable from its respective main memory binary module's (or module in packed format or OFP load format) embedded signature.

3.2 Using a light-weight processing node for a bridging multi-block binary Merkle tree proof in operational flight program authentication

The authentication of the OFP requires verification against the blockchain, but it is not always practical to have a copy of the entire blockchain, either due to storage limitations or due to limited communication bandwidth. For the sake of practicality, there is a need for a remote verification capability or Light-Weight Processing Node (LWPN) (Bitcoin Wiki, 2018a) to process a small part of the blockchain, which will authenticate an input OFP at or near, the endpoint. From the block's root hash, with a limited number of selected hashes at various levels about the tree, one could verify a specific transaction without having to check every transaction hash in the block of the transaction of interest. This greatly reduces the amount of information required for authentication.

For a particular upgrade OFP source code module, an LWPN approach can be used to authenticate the upgrade OFP such that the whole blockchain is not needed for authentication, but rather, a Merkle proof based on a small set of blockchain-derived data is needed. In consideration of the above, the authors present the Bridging Multi-block PBMT (BMP).

There is an information storage economy with the PBMT in that for a receiving LWPN, it would require $\log_2 n$ provided hashes with $\log_2 n + 1$ hashing operations to prove a specific transaction in an n transaction containing block to confirm the corresponding root hash. For example, if there are from the previous section, about two thousand transactions per block $\approx 2^{11} = 2048$, a Merkle proof only requires 12 hashing operations with 11 sender-provided hashes, such that a party requiring verification can verify that a specific transaction is part of a recognized blockchain block, with the caveat that the party requiring verification has a separate channel that can access the blockchain root hash independent from the sender. Otherwise, a malicious sender might send fabricated blockchain data.

The application of the PBMT is illustrated in Figure 8, where the endpoint branch of the BMP is shown. From the SSA, a Merkle tree is generated containing magenta nodes with regard to the upgraded binary module, shown in orange, including the high-level root hash for both systems. This is the SSA branch of the BMP. When combined and calculated up to the root hash, this is enough to prove that the particular OFP upgrade binary module, whose corresponding source code is derived from its ancestral source code of the previous version, is legitimate. The previous version will have a corresponding binary module in the OFP currently resident on the endpoint. This is consistent with the nature of embedded software in many avionics systems whereby source code gradually evolves over the lifecycle of its corresponding avionic system endpoint target.

From the endpoint, using information stored at or around, the endpoint, a Merkle tree representation is generated containing green nodes with regard to the endpoint resident binary section derived from its respective source code, shown in blue. The Merkle proof

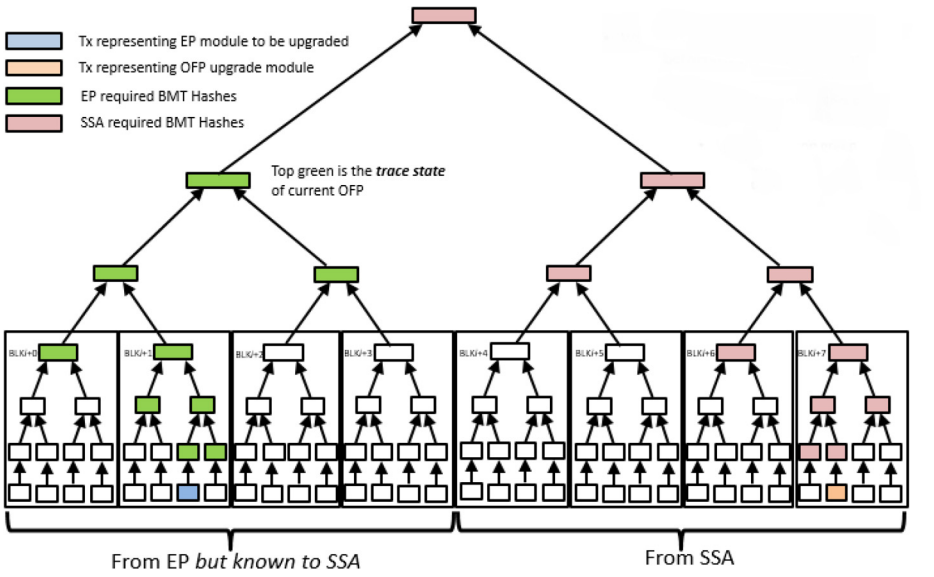

Figure 8.
Illustration of a bridging multi-block PBMT (BMP) that represents a Merkle proof

shown is that a newer version of source code (orange) is derived from an older source code (blue) from the endpoint branch securely stored on the endpoint (green) and the SSA branch and root hash provided by the SSA with the OFP (magenta).

The OFP upgrade is associated with the section of blockchain-derived data (the magenta SSA branch in Figure 8) that relates to other blockchain-derived data stored at or near, the endpoint (the green endpoint branch).

To summarize, in the PARANOID authentication scheme associated with an OFP upgrade, the SSA provides a BMP SSA branch and root hash for the OFP upgrade so that, together with the endpoint's BMP branch, it can be used to bridge the blockchain traceability from the currently loaded OFP binary module resident on the avionics endpoint to the OFP corresponding upgrade binary module, therefore establishing correspondence for a specific endpoint binary module *vis-à-vis* its corresponding upgrade OFP binary module. This authenticates the upgrade OFP in a source code module-wise manner. This is a module-wise Merkle Proof of authenticity. The set of blockchain data to support the Merkle proof would be stored securely around the point of authentication to prevent a maliciously modified OFP, with an associated fabricated blockchain, from being injected into the said point of authentication.

What matters is that the final edit transaction involving the source file of interest contained in the last OFP and the final edit transaction of the corresponding source code file contained in the newer OFP are the lowest level leaf nodes in the BMP. Upon completion of development for a software upgrade cycle for a particular file, on the upgrade, there is the final version, *i.e.* the final edit, that corresponds final version on the currently fielded OFP.

Recapitulating the order of traceability: OFP format → packed storage format → main memory binary image → object file → source code. Having a BMP for each source code file (module) corresponding to a function in the object import list with regard to the linking operation (Samek, 2018), the blockchain-based authentication strategy would look like this: A securely implemented checking process would sequentially sweep through the upgrade OFP, whereby each binary module (resulting from an associated source code module [file]) with its specific embedded signature block as shown in Figure 6, would be processed against the BMP, likewise this would occur with the previous OFP, such that for the corresponding sections, based on the associated BMP, it could be proven module-wise, as in Figure 8, that the upgrade OFP is derived from the previous OFP.

3.3 Estimating the amount of blockchain derived data required to support OFP authentication

Under PARANOID, an authentication data set would be provided by the SSA for authentication at or near, the target endpoint. A Rough Order of Magnitude (ROM) size for the total amount of authentication information required by PARANOID to authenticate a typical naval avionics OFP, consisting of thousands of source code modules, is on the order of five million lines of code (LoC), including the .h include files of a C/C^{++} code base. For the sake of example, consider that, on average, there are 50 LoC in each source module (roughly a page of 10 point font) in the standalone source code file. This results in 100,000 files containing all the source code in the codebase.

Consider the issue of relating the source code of the currently loaded OFP with the source code of the upgrade OFP from blockchain data representing source code edits. This involves the 100,000 source could module files a build implemented under PARANOID, therefore, needs to provide proof for 100,000 development transactions representing the final development transaction for each source-code module. Now, consider the edit history of any given source module. There could be many edits (commits) between the source code module of the previous OFP and its corresponding upgrade version. For the sake of this example, let

the hypothetical blockchain used for this calculation example be similar to Bitcoin in transaction size, block size and total size. As of the writing of this paper is approaching one-half billion transactions (Blockchain Luxembourg S.A., 2020c).

Each transaction in Bitcoin consists of approximately 250 bytes (Stack Exchange Inc, 2014; Bitcoin Wiki, 2018d) and involves an SHA-256 hash, a public key infrastructure (PKI) signature, public key, source and destination addresses corresponding to the transaction parties. Let us consider a blockchain system representing one-half billion transactions. We will relate this to Figure 8 for calculating the number of required hashes over the entire set of transaction records in the one-half billion records, from the EP branch and the SSA. For BMPs, each leaf is a hash, *e.g.* SHA-256, so it would contain 32 bytes. So, the size of the BMP of a blockchain containing n transactions corresponds to $\log_2 n$, where n=one-half billion. For the SSA magenta branch this would be the amount of information is $\log_2$(one-half billion) $\times$ 32 Bytes $\approx 29 \times 32$ Bytes = 928 bytes $\approx$ 1kilobyte per BMP. For the green EP side there would need to be $28 \times 32 = 896 \approx$ bytes so that the total is about 2 kilobytes per BMP. Therefore, for an OFP with 100,000 source code (module) files, this results in a size of 200 megabytes. The previous paragraph describes the blockchain process of relating source code of the currently loaded OFP with the upgrade OFP source code, however, before that can be accomplished, one also must relate any OFP format module to its corresponding source code. Here we discuss the the requirements for the latter. Consider the transactions representing: editing, compilation, linking (build, image), packed mass memory storage format and OFP encoding format. From the previous section 3.2 regarding the 2,000 transaction record blocksize, within a block, there are approximately $10 \times \log_2(2048)-1 = 10$ hashes where each hash is 32 bytes. For the 100,000 modules, we relate each transaction involving edit, compile, link, packing and OFP format encoding to its blockchain root hash via the Merkle proof discussed in section 3.2. The total number of bytes, therefore, is $10 \times 32 \times 5 \times 100{,}000 = 162$ million bytes. So, approximately 200 + 160 = 360 megabytes, would be needed to store the blockchain derived information.

This may appear to be an exorbitantly large amount of information for supporting authentication. It is considerably more information than what is required for a corresponding hash or HMAC. However, remember that this is authenticating *the entire history of the code* down to individual modules, proving every section of code is TECHS compliant over its development history, including the toolchain used in code development.

Considering the latest storage technology, this is not an impractically large amount of data. As of the time of writing, a typical high-end hard disk drive (HDD) can hold approximately 15 terabytes (on the order of 75 Bitcoin blockchains). High-end solid-state drives (SDDs), more likely to be in a militarized platform, are also in this capacity range. Intel Optane (Intel Corporation, 2019b) memory provides 32 gigabytes of non-volatile memory in a module package of only 22 millimeters by 80 millimeters, already having a significantly higher density than flash technology used in conventional SSDs. In the unlikely event that It is required to use something the size of the entire Bitcoin blockchain, a modern high-end drive would be able to support it. As of this writing, avionics single-board computers typically have a minimum capacity of multiple gigabytes of non-volatile memory (Curtiss-Wright, 2018). Several hundred megabytes would be easily supportable even with conventional technology.

One may also note that the cryptography community is moving in the direction of more elaborate crypto-authentication schemes, per National Security Agency (NSA) direction, in reaction to the possibility of quantum computing that could break presently used PKI encryption schemes, such that an adversary could forge signatures and HMAC hashes (Chen, 2006). A significant characteristic of post-quantum schemes is that, by orders of

magnitude, much larger amounts of keying information is required for signatures and public keys (Bernstein *et al.*, 2009). Thus, in the future, significantly larger storage requirements for cryptographic data are inevitable. For full post-quantum capability, for PARANOID and other blockchain-based applications, the legacy PKI signature portion of some blockchain protocols would have to be replaced by post-quantum cryptography signature algorithms such as lattice cryptography schemes, *e.g.* The NTRU Signature Algorithm (NTRUSign) (Institute of Electrical and Electronics Engineers, 2008). NIST has held competitions for post-quantum cryptographic algorithms including signature algorithms (National Institute of Standards and Technology, 2018).

Another matter to consider is how long it will take to process the authentication data. As the majority of processing involves hashing, *e.g.* SHA-2, it would typically not take very long. Intel Corporation has special processor instructions to accelerate hashing by combining certain steps of the SHA-2 and SHA-3 algorithms into single instructions for the x86 and x86-64 processor architectures (Guilford *et al.*, 2012; Intel Corporation, 2016a). In benchmarks of the SHA-2 (SHA-256) and SHA-3 implementation, x86-64 hardware has been shown to be able to process over 100 megabytes per second (Intel Corporation, 2016a; wolfSSL, Inc, 2018). Then, there is also the matter of signature verification. The website wolfSSL (wolfSSL, Inc, 2018) reports that over 1,600 Elliptical Curve Digital Signature Algorithm (ECDSA) (National Institute of Standards and Technology, 2013) signature verifications can be performed per second. ECDSA is the specific signature algorithm used in Bitcoin and other cryptocurrencies.

3.4 Securing the operational flight program through the program loader

The long-term goal should be to provide the endpoint the ability to detect any unapproved binaries. PARANOID would allow the avionics endpoint to authenticate its OFP software binary code. The endpoint might perform a simple software-based Merkle proof per avionics board inside a WRA, or per avionics unit applied to the packed binary at startup prior to unpacking the packed image stored in nonvolatile memory. This capability would require the set of BMPs for the specific SBC binary target unit corresponding to that particular avionics SBC. Storing this data would require a secure region of memory. The secure memory would relate to "sealed storage" technology and may take advantage of system hardware security features. The specific software binary for implementing the Merkle proof authentication function would also need to be carefully protected.

An even more ambitious approach may be to perform a continuous Merkle proof applied to the main memory image of the loaded binary running in the background and implemented by the main memory hardware such as the Memory Management Unit (MMU) (Uyeda, 2009), continuously checking the main memory image of the code. A secure way to perform this proof would have to be developed. There have been researching efforts to develop high-security MMU systems (Gilmont *et al.*, 1999) that may have some capability to support PARANOID, but do not yet exist in available products. Though it may be possible to have a low-level hardware system performing Merkle proofs implemented by the MMU, this may be difficult to implement on existing avionics due to the requisite amount of modification and, thus, may have to be relegated to future systems.

Considering the difficulty of the above endpoint authentication with regard to the necessity for developing new, specific MMU hardware to support it, the next logical place to perform PARANOID-based authentication would be at the point of loading the OFP into the aircraft.

One could imagine a ruggedized computer with state-of-the-art robust security and a production secure VM running on top of a secure hypervisor that manages the OFP load process. It could even be the identical SDE developed under PARANOID with the development functions disabled but retaining the PARANOID authentication functionality. The SDE would do a section-by-section authentication of the OFP as it was being accessed for transfer to the target avionics system in the manner described in Section 3.1. With this approach, every TECHS-related aspect of the OFP can be authenticated. For this capability, it may be appropriate to create an extension of MIL-STD-2217 that adds a blockchain-based authentication capability to the load process. This approach to performing blockchain-based authentication at the point of load would be much more achievable in the near term, allowing the continued use of legacy avionics but increasing the ability to have robust authentication because of the available technology to support its implementation.

Consider PC-based loaders hosting a secure VM-based system, for instance. The two existing OFP loader systems used in the Navy today are the legacy Memory Loader Verifier Set (MLVS) AN/USQ-131A and B, and the newer Program Loader Set (PLS). The existing load process already has some capability with regard to the detection of OFP corruption. MIL-STD-2217 describes a standard for loading an OFP onto an avionics target via a serial data bus connection while simultaneously performing verification (United States Department of Defense, 1991).

As avionics WRAs may consist of multiple SBCs, each executing its own specific binary image, all the various binaries targeting specific SBCs across a WRA in an avionics system are bundled together to form an OFP for a specific WRA. Usually, a unique SBC uses a specific binary image, but sometimes a specific binary image may be used across multiple identical SBCs and sometimes, multiple identical SBCs each get a unique binary image. For this purpose, the generated OFP is a file that consolidates multiple binary images into a special format containing headers that describe to which target SBCs the various images go. The OFPs are distributed to naval squadron hosts supporting avionics and are subsequently loaded onto the program loaders.

The complete set of interoperable OFPs supporting a set of WRAs in an aircraft avionics system is a Software Compatibility Set (SCS). The SCS for a specific avionics software version is loaded onto a program loader as received from the SSA. For PARANOID, generating the OFP load format from its constituent images could be viewed as a critical development function involving its particular stage of signing and blockchain of outputs. The signed OFPs for a particular SCS could have embedded labels indicating the SCS version number as added security.

4. Other considerations to maximize the benefits of the PARANOID approach

4.1 Networking on a PARANOID system

Network connections could be implemented via a secure Virtual Private Network (VPN). The networking would allow SDEs across multiple development hosts to communicate confidentiality between the hosts. Major builds could be implemented across multiple SDEs, perhaps, using a "director" SDE that contains a build script written by a lead developer. Another notional architecture is a system whereby individual developers have workstations with SDEs that use the blockchain-enabled editors for the purpose of writing source code. The workstations could be connected to a build server running an SDE that performs compilations and system builds. Thus, PARANOID would be compatible with a networked development and operations (DevOps) system.

4.2 Limit scope of developers' specifically-assigned areas

The assignment process, as described in Section 2.5, might also be useful for limiting the scope of what individual developers can work on. This would help mitigate malicious insider threats. For instance, if a particular developer is assigned to write code to support a navigation system, perhaps, that specific developer should not have access to any code that supports the fire control system. Smart contracts might be used to create restricted lines of development, whereby a coder assigned to one line of code development does not have access to other lines. The restriction process might be implemented anywhere on the chain of authority, but might be most appropriately done at the supervisor level. Alternate variants of TADA could be created at the HLA, each corresponding to different embedded development areas, *e.g.* fire control software versus navigation software. The TCM might be used to indicate stovepiped restrictions on TADA, as described in Section 2.6.

4.3 Enforce specifications with formal proofs and blockchain

In a similar vein, some specifications of the software might be written by the supervisor as part of the assignment process. Today, some coding is based on specification languages (Sennella and Wirsing, 2019) such as the Specification and Description Language (SDL) (SDL-RT, 2013) or Gallina (Coq, 2020); or formal languages such as ML (Github Project: ML, 2020), Haskell (Haskell, 2020) or Occam (Occam 2.1, 2020). The above specification language code is implemented by a "satisfiability solver" (Gomes, 2008) or proof systems such as Coq (The Coq Proof Assistant, 2020), HOL (Higher Order Logic Interactive Theorem Prover, 2020) or CSP (Hoare, 2015). The supervisor might be able to write a specification that limits what the code might do in regard to some basic attributes. For example, for some particular source code representing a function, a supervisor could limit the input and output arguments or specific, more complex specifications. The assigned developer could then write the main body of the code compliant with the specification, as verified by formal proof. The formal proof would become a critical development function, which would be represented by a transaction. It could be determined that a binary conforms to a formal proof without the specification being publically released.

4.4 Use of blockchain to monitor access to sensitive code or sensitive data

If the software is sensitive, once the software has been developed and the code is "frozen" by the appropriate level of management, the priority concern may become protecting the software against unauthorized disclosure. A big part of dealing with this concern is monitoring who has access to the software to prevent the exfiltration of sensitive information. Making each access attempt a transaction recorded on a blockchain, using a method similar to what has been previously presented, could enable tracking of personnel access to sensitive software, documentation or data.

A controlled access blockchain (CABlock) is a secure distributed access control system that is built on a blockchain infrastructure to provide a transactional approach to implementing access to sensitive software, documentation and/or data. Additionally, it enforces the need-to-know access restrictions on IT support staff, system administrators and security personnel, who may not need access to the actual data and documents, to ensure and enforce security (Di-Francesco Maesa *et al.*, 2017). This supports the concept of Role-Based Access Control (RBAC). Security personnel can audit a blockchain ledger representing transactions by administrators and program office personnel to support administrative functions. However, such transactions do not need to involve read access to view the actual files. This fundamental limit would itself be represented in the ledger as records of granted permissions pertaining to defined access functions or query functions

that, depending on the particular user's clearance and need-to-know status, provide limited information not involving direct read access. This is a variation of zero-knowledge proof (Schor, 2018). Some aspect about a secret is proven without transferring complete knowledge of the secret itself. The above functions would be implemented through a secure system such as a secure VM in the manner of the SDE. The execution of such a function would be implemented as a transaction creating a blockchain record representing what specific information was accessed when the access occurred and who was given access and any other necessary information of interest to the delegated authority. The aforementioned access functions would be enabled or disabled by the permissions records contained in a block but could be repudiated with a fork or a new block. A TADA system could also be used to enable permissions for a limited time corresponding to a defined task assignment period or work-year. Different classes of TADA might be provided simultaneously to enable classes of permissions assigned to specific users according to each user's specific clearance level and need-to-know status *vis-à-vis* the sensitivity of the material.

A blockchain can support the secure storage of any kind of data, including documents, models and code with the optional addition of smart contracts that would affect access by releasing decryption keys to designated individuals once required conditions have been met. This novel implementation is mainly focused on access control and access tracking of sensitive material. The key to the success of a blockchain is that it is not limited to one network administrator or to one single computer. A host of nodes are used to share the data and enable the privileged end-users with high integrity on the nodes. The strength of a blockchain is the recognition that ownership or control, of an asset, is determined by the consensus of these nodes. Additionally, with the confidentiality, integrity and availability aspect of the encrypted and authenticated blockchain, it is very difficult to insert malicious data and/or viruses. A document control list (like a history) would be found for each document in the ledger. This would provide a clear mechanism for forensic analysis in the event of a mishap and provide immediate notification of an incident to participating nodes. Finally, rules can be applied to a blockchain that prevents viewing documents outside of the private network and enforces rules such as document self-destruct.

4.5 Limits of PARANOID

It should be noted that this approach does not directly prevent certain issues such as zero-day vulnerabilities. As a zero-day is a failure of a developer to foresee how code might be compromised, the ability to automatically detect zero-day vulnerabilities is beyond the core capability and scope of PARANOID with regard to enforcing the TECHS condition. PARANOID may support, however, the detection of zero-days by enforcing peer review and the application of static analyzers as critical development stages to provide a greater likelihood of detection (*i.e.* use of these would also have utility for detecting a malicious insider). Once a zero-day is detected, smart contracts-based revocation in conjunction with blockchain-based authentication at the point of load might be used to prevent the affected OFP containing the specific zero-day-affected components from being loaded onto an endpoint.

Finally, PARANOID uses the consensus mechanisms of existing blockchain systems. It would be up to the implementers to select a blockchain system with a robust consensus algorithm that would minimize the likelihood of a 51% attack. Using a private, permissioned blockchain, as opposed to a public blockchain should also contribute to robustness against a 51% attack.

4.6 Ongoing proof-of-concept demonstration

A team has been stood up within the Naval Air Warfare Center Aircraft Division (NAWCAD) to implement a proof-of-concept demonstration of PARANOID based on the open-source Hyperledger Fabric blockchain architecture (Androulaki *et al.*, 2018).

At the time of this writing, the PARANOID project team is proceeding to create a proof-of-concept prototype to demonstrate how the previously-described DevSecOps approach can be implemented into a workable system. For editing software operating under PARANOID, the team created a fork of the open-source text editor Nano, which is written in C and designed and configured several other components that would then be integrated into Nano by modifying its source code base to create new program logic and to interface functional components. The components integrated into Nano are: 1) a Git repository interface to provide source control of a developer's project source code and configuration files; and 2) multiple Hyperledger chain code definitions to support source code commit transactions (*i.e.* using Git to manage the source code files and Hyperledger to securely store the cryptographic hash of each file to validate the Git repository). Additionally, a compile transaction has been implemented for a blockchain-enabled version of the open-source GNU GCC compiler to represent a compilation of source files having edit histories that are themselves blockchain represented. In a similar manner, a link transaction has been implemented for the GNU linker regarding linking operation on blockchain-represented object files. Because every transaction in PARANOID involves exchanging TADA, the team created a Hyperledger chaincode that manages and tracks TADA expenditures for the aforementioned transactions.

The upper left of Figure 9 is an example of how a file edit is stored on the blockchain over time. When querying a file's history, it will print out from newest edits to oldest edits. In the upper right of Figure 9, there is an example of a PARANOID log file generated during runtime. The lower left of Figure 9 shows an example of embedded information in the resulting executable file from the PARANOID linking process. Note that the local directory path information has been removed. Finally, in the lower right of Figure 9 is an illustration of what a user will see in Nano as the user is interacting with PARANOID blockchain commands.

In the proof-of-concept, SDEs that implement PARANOID run in Docker containers. Intel SGX will be used to attest that any particular SDE is free from unauthorized modifications at the point of performing a development function. The attestation will be integrated into the development transaction message dispatched to the blockchain network. Currently, the

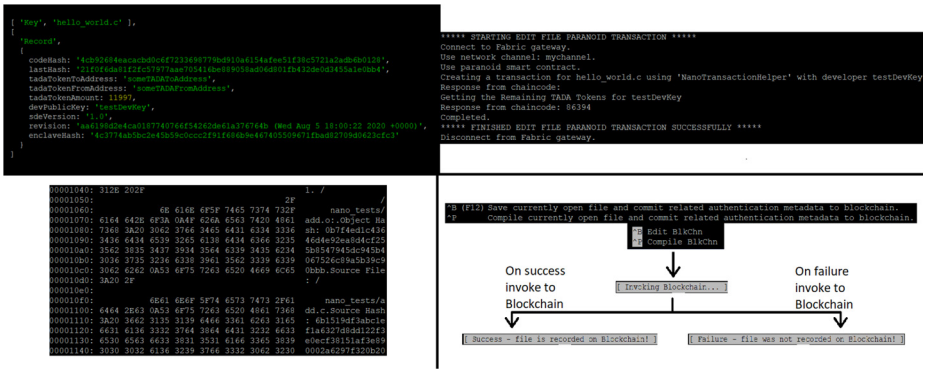

Figure 9. Screenshots from the PARANOID proof-of-concept system currently under development

authors are investigating a solution for applying SGX enclave security at the Docker container level with products such as SCONE (Arnautov *et al.*, 2016) or Graphene (Tsai, 2017). The project team plans to finish the development of the PARANOID proof-of-concept such that it might be tested in a development cloud associated with Naval Air Systems Command (NAVAIR) aircraft programs. Here, system scalability will be characterized to determine the amount of blockchain processing resources that would be required to support a typical size naval aircraft avionics software development program. The ability to perform a blockchain-based authentication at the point of loading a simulated "dummy" OFP on a simulated endpoint will also be demonstrated.

In the near future, as described in Section 2.2.2, the SDE can be used to develop itself in a self-hosted manner such that an SDE can modify itself and recompile itself where the development history of every SDE software component is represented on a blockchain. Thus the blockchain-based traceability extends from the OFP embedded software over its history to the toolchain used to develop the embedded software over said toolchain's development history. At this point, PARANOID might become a significant undertaking with the need to support software requirements across multiple programs, and thus, might require a maintainer/developer entity, also mentioned in Section 2.2.2.

4.7 Applicability of PARANOID to commercial embedded development

The approach described herein can also be applied to commercial systems. One obvious application is too embedded software development for commercial aviation systems. Autonomous vehicles are another potential application area of this approach. As automation advances, the emphasis moves from a human driver to a collection of hardware and software that define the driver. As such, many control manufacturers represent the key ingredients of an autonomous vehicle to include safety-critical embedded software associated with LiDAR, sonar, radar, imaging and machine learning systems. The authors posit that regulators and consumers will ultimately appreciate an integrated approach to holistically securing the autonomous vehicle's software supply chain in a manner in which a software program load is traceable and provably developed and built with authorized tools on an authorized DevOps system by authorized developers, based on a universally-accepted blockchain. An example may be software authenticated and passed from control manufacturers to carmakers and then subsequently to service providers for software maintenance and upgrades. Still, other potential commercial application areas might be embedded software for medical devices and medical equipment or any software product that has critical safety and/or security requirements in areas of transportation, industrial controls, communications and finance.

5. Conclusion

The authors have presented a method for implementing strong, fine-grained authentication and traceability to the SSA to support OFP security in naval avionics. While the concept is strongly based on blockchain technology, considerations for the SDE, as presented, are paramount for the practical implementation of this method. The authors posit that the strength of the approach is the application of a transactional paradigm, in an economic sense, involving expenditure and a receipt in the form of a blockchain transaction. The resulting software development method, with the judicious combination of blockchain and host-based security technology, emphasizing signing and embedded signature for critical development functions, can successfully meet the increasing cybersecurity challenges faced by the US armed forces. A plethora of developing technology can be used to implement this vision, from open source technology to products that specifically support blockchain applications.

Note

1. A smart contract is a "cryptographic 'box' that contains value and only unlocks if certain conditions are met" (Buterin, 2014a). In this case it is a created code that exists on the blockchain, which proves that a function has been completed.

References

Anderson, E.A., Irvine, C.E. and Schell, R.R. (2004), "Subversion as a threat in information warfare", Space and Naval Warfare Systems Center North Charleston, SC, available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.505&rep=rep1&type=pdf (accessed 13 January 2020).

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y. and Muralidharan, S. (2018), "Hyperledger fabric: a distributed operating system for permissioned blockchains", EuroSys '18, April 23-26, 2018, Porto, available at: https://arxiv.org/abs/1801.10228 (accessed 13 January 2020).

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'keeffe, D., Stillwell, M.L. and Goltzsche, D. (2016), "SCONE: secure Linux containers with intel SGX", November 2–4, USAISBN 978-1-931971-33-1, Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems, Savannah, GA.

Atlassian (2018), "Jira homepage", available at: www.atlassian.com/software/jira (accessed 13 January 2020).

Bernstein, D.J., Buchmann, J. and Dahmen, E. (Eds) (2009), *Post-Quantum Cryptography*, Place of Publication, Springer-Verlag Berlin Heidelberg, available at: www.springer.com/gp/book/9783540887010 (accessed 17 January 2020).

Bitcoin Wiki (2018a), "Lightweight node", available at: https://en.bitcoin.it/wiki/Lightweight_node (accessed 13 January 2020).

Bitcoin Wiki (2018b), "Proof of burn", available at: https://en.bitcoin.it/wiki/Proof_of_Burn (accessed 13 January 2020).

Bitcoin Wiki (2018c), "Proof of stake", available at: https://en.bitcoin.it/wiki/Proof_of_Stake (accessed 13 January 2020).

Bitcoin Wiki (2018d), "Transaction", available at: https://en.bitcoin.it/wiki/Transaction (accessed 13 January 2020).

Blockchain Luxembourg S.A (2020a), "Average transactions per block", available at: www.blockchain.com/charts/n-transactions-per-block?timespan=2years (accessed 4 October 2020).

Blockchain Luxembourg S.A (2020b), "Block size", available at: www.blockchain.com/charts/avg-block-size (accessed 4 October 2020).

Blockchain Luxembourg S.A (2020c), "Total number of transactions", available at: www.blockchain.com/charts/n-transactions-total (accessed 4 October 2020).

Brown, S. and Hurt, T. (2017), "Engineering software assurance into weapons systems during the DoD acquisition life cycle", *Journal of Cyber Security and Information Systems*, Vol. 5 No. 3, available at: www.csiac.org/journal-article/engineering-software-assurance-into-weapons-systems-during-the-dod-acquisition-life-cycle/ (accessed 17 January 2020).

Bryson, D., Penny, D., Goldberg, D.C. and Serrao, G. (2017), "Blockchain technology for government", Technical report, The MITRE Corporation, available at: www.mitre.org/sites/default/files/publications/blockchain-technology-for-government-18-1069.pdf (accessed 13 January 2020).

Burton, G. (2017), "Software maker admits attackers hid backdoor in entire suite of products", available at: www.computing.co.uk/ctg/news/3015824/software-maker-admits-attackers-hid-backdoor-in-entire-suite-of-products (accessed 7 January 2020).

Buterin, V. (2014a), "A next-generation smart contract and decentralized application platform", available at: http://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf (accessed 13 January 2020).

Buterin, V. (2014b), "On stake", Ethereum Blog 5, available at: https://blog.ethereum.org/2014/07/05/stake/ (accessed 13 January 2020).

Buterin, V. (2014c), "Slasher: a punitive proof-of-stake algorithm", Ethereum Blog, available at: https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm (accessed 13 January 2020).

Buterin, V. (2014d), "Slasher ghost, and other developments in proof of stake", Ethereum Foundation 3 (accessed 13 January 2020).

Buterin, V. (2016), "DEVCON1: understanding the ethereum blockchain protocol – Vitalik buterin", available at: www.youtube.com/watch?v=gjwr-7PgpN8 (accessed 29 January 2020).

CBS (2015), "The spy among Us, parts I and II", 60 Minutes, CBS News 10 May 2015, available at: www.cbsnews.com/news/former-kgb-spy-jack-barsky-steve-kroft-60-minutes/ (accessed 7 January 2020).

Chen, L. (2006), "NISTIR 8105, report on post-quantum cryptography", NIST, available at: https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf (accessed 13 January 2020).

Collier, C.P. (2016), "Hardware open systems technology (HOST)", The Open Group/NAVAIR PMA-209, available at: www.embeddedtechtrends.com/2017/PDF_Presentations/T11A%20-%20HOST.pdf (accessed 7 January 2020).

Constantin, L. (2017), "What is a supply chain attack", Motherboard, available at: https://motherboard.vice.com/en_us/article/d3y48v/what-is-a-supply-chain-attack (accessed 7 January 2020).

Coombs, R. (2015), "Securing the future of authentication with ARM TrustZone-based trusted execution environment and fast identity", ARM white paper, May 25.

Coq (2020), "Proof assistant online reference manual: Gallina specification language", available at: https://coq.inria.fr/refman/language/gallina-specification-language.html (accessed 17 January 2020).

Curtiss-Wright (2018), "VPX6-1958 6U VPX intel core i7 Haswell SBC", available at: www.curtisswrightds.com/products/cots-boards/processor-cards/6u-intel-sbc/vpx6-1958.html (accessed 13 January 2020).

Defense Human Resource Activity (2018), "CAC: DoD common access card", available at: www.cac.mil (accessed 13 January 2020).

Di-Francesco Maesa, D., Mori, P. and Ricci, L. (2017), "Blockchain based access control", *Distributed Applications and Interoperable Systems*, in Indulska, J. and Raymond, K. (Eds), Place of Publication, Springer-Verlag, Berlin Heidelberg, available at: www.springer.com/gp/book/9783540728818 (accessed 29 January 2020).

Digiconomist (2018), "Bitcoin energy consumption index", available at: https://digiconomist.net/bitcoin-energy-consumption (accessed 13 January 2020).

Dwork, C. and Nair, M. (1992), "Pricing via processing or combatting junk mail", *CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology August 1992*, pp. 139-147, available at: https://dl.acm.org/doi/10.5555/646757.705669 (accessed 13 January 2020).

Gegick, M. and Barnum, S. (2005), "'Reluctance to trust', United States computer emergency response team (US-CERT)", available at: www.us-cert.gov/bsi/articles/ knowledge/principles/reluctance-to-trust (accessed 13 January 2020).

git webpage (2019), available at: https://git-scm.com/ (accessed 13 January 2020).

Github Project: ML (2020), available at: http://sml-family.org/ (accessed 17 January 2020).

GlobalPlatform (2008), *Trusted Execution Environment System Architecture v1.1,* available at: https://globalplatform.org/specs-library/?filter-committee=tee#collapse-1

Gilmont, T., Legat, J.-D. and Quisquater, J-J. (1999), "Enhancing security in the memory management unit", *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium, IEEE*, Vol. 1.

Gomes, C.P., Kautz, H., Sabharwal, A. and Selman, B. (2008), Chapter 2, "Satisfiability solvers", in Van Harmelen, F., Lifschitz, V. and Porter, B. (Eds), *Handbook of Knowledge Representations*, Elsevier, available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.399.4739&rep=rep1&type=pdf (accessed 17 January 2020).

GrammaTech, Inc (2018), "GrammaTech homepage", available at: https://grammatech.com (accessed 7 January 2020).

GReAT (2017), "ShadowPad in corporate networks: popular server management software hit in supply chain attack", Kaspersky Lab, available at: https://securelist.com/shadowpad-in-corporate-networks/81432/ (accessed 7 January 2020).

Guilford, J. Kirk, Y. and Vinodh, G. (2012), "Fast SHA-256 implementations on intel architecture processors", available at: www.intel.com/content/dam/www/public/us/en/documents/white-papers/sha-256-implementations-paper.pdf (accessed 13 January 2020).

Haskell (2020), available at: www.haskell.org/ (accessed 17 January 2020).

Higgins, K.J. (2018), "Chinese APT backdoor found in CCleaner supply chain attack", DARKReading, available at: www.darkreading.com/endpoint/privacy/chinese-apt-backdoor-found-in-ccleaner-supply-chain-attack/d/d-id/1331250 (accessed 7 January 2020).

Higher Order Logic Interactive Theorem Prover (2020), available at: https://hol-theorem-prover.org/ (accessed 4 October 2020).

Hoare, C.A.R. (2015), "Communicating sequential processes", May 18, 2015, available at: https://dl.acm.org/doi/book/10.5555/3921 (accessed 17 January 2020).

Idialu, K. (2019), "Ellcrys technical white paper, (V1)", available at: https://github.com/ellcrys/papers (accessed 13 January 2020).

Institute of Electrical and Electronics Engineers (2008), "IEEE 1363.1-2008 – IEEE standard specification for public key cryptographic techniques based on hard problems over lattices", available at: https://standards.ieee.org/standard/1363_1-2008.html (accessed 13 January 2020).

Intel Corporation (2014), "Software guard extensions programming reference", 329298-002US, available at: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf (accessed 7 January 2020).

Intel Corporation (2016a), "Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4", available at: https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4 (accessed 13 January 2020).

Intel Corporation (2016b), "Local (Intra-Platform) attestation", Intel Developer Zone, available at: https://software.intel.com/en-us/node/702983 (accessed 8 January 2020).

Intel Corporation (2019a), "Trusted execution technology (intel® TXT)", Software Development Guide: Measured Launch Environment Development Guide, Rev 016 (accessed 7 January 2020).

Intel Corporation (2019b), "Optane technology video on the intel website", available at: www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html (accessed 13 January 2020).

Jakobsson, M. and Juels, A. (1999), "Proofs of work and bread pudding protocols", Communications and Multimedia Security, Kluwer Academic Publishers, pp. 258-272, available at: www.semanticscholar.org/paper/Proofs-of-Work-and-Bread-Pudding-Protocols-Jakobsson-Juels/1745e5dbdeb4575c6f8376c9e75e70650a7e2e29 (accessed 13 January 2020).

King, S. and Nadal, S. (2012), "Ppcoin: peer-to-peer crypto-currency with proof-of-stake", self-published paper, available at: https://decred.org/research/king2012.pdf (accessed 13 January 2020).

Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R. and Heiser, G. (2014), "Comprehensive formal verification of an OS MicroKernel", *ACM Transactions on Computer Systems*, Vol. 32 No. 1, Article 2, available at: https://dl.acm.org/doi/10.1145/2560537 (accessed 8 January 2020).

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M. and Sewell, T. (2009), "seL4: formal verification of an OS kernel", *22nd Association for Computing Machinery (ACM) Symposium on Principles of Operating Systems*, available at: https://dl.acm.org/doi/10.1145/1629575.1629596 (accessed 8 January 2020).

Lamport, L., Shostak, R. and Pease, M. (1982), "The byzantine generals problem", *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 3, pp. 382-401, available at: https://dl.acm.org/doi/10.1145/357172.357176 (accessed 8 January 2020).

Linux Foundation, Tool Interface Standard Committee (1995), "Tool interface standard (TIS) executable and linking format (ELF) specification", available at: https://refspecs.linuxbase.org/elf/elf.pdf (accessed 4 October 2020).

Nakamoto, S. (2008), "Bitcoin: a peer-to-peer electronic cash system", available at: www.researchgate.net/publication/228640975_Bitcoin_A_Peer-to-Peer_Electronic_Cash_System (accessed 13 January 2020).

National Institute of Standards and Technology (1994), "Guideline for the use of advanced authentication technology alternatives (FIPS-190)", available at: https://csrc.nist.gov/publications/detail/fips/190/archive/1994-09-28 (accessed 13 January 2020).

National Institute of Standards and Technology (2008), "The keyed- hash message authentication code (HMAC) (FIPS-198-1)", Computer Security Resource Center (CSRC), available at: https://csrc.nist.gov/publications/detail/fips/198/1/final (accessed 7 January 2020).

National Institute of Standards and Technology (2013), "Federal information processing standards publication, digital signature standard (FIPS-186-4)", available at: https://csrc.nist.gov/publications/detail/fips/186/4/final (accessed 13 January 2020).

National Institute of Standards and Technology (2015), "FIPS-180-4 secure hash standard", Computer Security Resource Center (CSRC), available at: https://csrc.nist.gov/publications/detail/fips/180/4/final (accessed 7 January 2020).

National Institute of Standards and Technology (2017), "Risk management framework for information systems and organizations", Draft NIST Special Publication, 800-37.

National Institute of Standards and Technology (2018), "NIST post-quantum cryptography standardization", available at: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

Occam 2.1 (2020), "(Programming language) reference manual", available at: www.wotug.org/occam/documentation/oc21refman.pdf (accessed 17 January 2020).

Platform One (2020), available at: https://software.af.mil/team/platformone/ (accessed 4 October 2020).

Rosic, A. (2018), "Crypto currency wallet guide: a step-by-step tutorial", available at: https://blockgeeks.com/guides/cryptocurrency-wallet-guide/ (accessed 13 January 2020).

Saarinen, J. (2017), "Security vendor Avast distributed malware-infested utility: CCleaner hacked", ITNews", available at: www.itnews.com.au/news/security-vendor-avast-distributed-malware-infested-utility-473590 (accessed 7 January 2020).

Samek, M. (2018), "Modern embedded system programming course, lesson 9", Quantum Leaps, LLC, available at: www.state-machine.com.

Schor, L. (2018), "On zero-knowledge proofs in blockchains", Medium, available at: https://medium.com/@argongroup/on-zero-knowledge-proofs-in-blockchains-14c48cfd1dd1 (accessed 17 January 2020).

SDL-RT (2013), "SDL-RT specification and description language – real time", available at: www.sdl-rt.org/standart/V2.3/pdf/SDL-RT.pdf (accessed 17 January 2020).

Secure Technology Alliance (2018), "Smart card technology FAQ", available at: www.securetechalliance.org/smart-cards-faqs (accessed 13 January 2020).

Sennella, D. and Wirsing, M. (2019), "8 specification languages", available at: https://pdfs.semanticscholar.org/c474/55200e0569f2592016d466b12b47a5fb6a21.pdf (accessed 17 January 2020).

Shah, S. (2017), "The INQUIRER", available at: www.theinquirer.net/inquirer/news/3017504/ccleaner-hack-supply-chain-attack-saw-app-install-malware-on-users-machines (accessed 7 January 2020).

Stack Exchange Inc (2014), "What is the average size of a bitcoin transaction?", Forum discussion, available at: https://bitcoin.stackexchange.com/questions/31974/what-is-the-average-size-of-a-bitcoin-transaction (accessed 13 January 2020).

Szabo, N. (1998), "Secure property titles with owner authority", available at: https://nakamotoinstitute.org/secure-property-titles/ (accessed 13 January 2020).

TCG (2020), "Trusted platform module (TPM) 1.2 main specification (2003)", available at: https://trustedcomputinggroup.org/resource/tpm-main-specification/ (accessed 7 January 2020).

The Coq Proof Assistant (2020), "Coq website", available at: https://coq.inria.fr/ (accessed 17 January 2020).

The MathWorks, Inc (2017), "10 Reasons to use static analysis for embedded software development", available at: www.mathworks.com/content/dam/mathworks/tag-team/Objects/p/93127v00_Polyspace_Static_Analysis_Whitepaper.pdf (accessed 8 January 2020).

The MITRE Corporation (2019), "Common weakness enumeration™ (CWE), list version 3.4.1", available at: https://cwe.mitre.org/data/index.html (accessed 7 January 2020).

The Open Group (2016), "Future airborne capability environment (FACE™) overview", REFERENCE: G165, available at: https://publications.opengroup.org/g165 (accessed 7 January 2020).

Thompson, K. (1984), "Reflections on trusting trust", *Communications of the ACM*, Vol. 27, No. 8.

Tsai, C.C., Porter, D. and Vij, M. (2017), "Graphene-SGX: a practical library OS for unmodified applications on SGX", *USENIX Annual Technical Conference*.

United States Department of Defense (1991), "MIL-STD-2217: memory loader/verifier multiplex bus interface with avionic systems", available at: https://quicksearch.dla.mil/qsDocDetails.aspx?ident_number=110630 (accessed 7 January 2020).

United States Department of Defense (2019), "DoD digital modernization strategy: DoD information resource management strategic plan FY 19-23", available at: https://media.defense.gov/2019/Jul/12/2002156622/-1/-1/1/DOD-DIGITAL-MODERNIZATION-STRATEGY-2019.PDF (accessed 7 January 2020).

Uyeda, F. (2009), "Lecture 7: memory management", CSE120, Principles of Operating Systems, available at: https://cseweb.ucsd.edu/classes/su09/cse120/lectures/Lecture7.pdf (accessed 29 January 2020).

Vasin, P. (2014), "Blackcoin's proof-of-stake protocol v2", available at: https://blackcoin.org/blackcoin-pos-protocol-v2-whitepaper.pdf (accessed 4 October 2020).

Wheeler, B. (2017), "Jack Barsky: the KGB spy who lived the American dream", BBC News, available at: www.bbc.com/news/magazine-38846022 (accessed 7 January 2020).

wolfSSL, Inc (2018), "Benchmarking wolfSSLl and wolfCrypt", available at: www.wolfssl.com/docs/benchmarks (accessed 13 January 2020).

World Crypto Index (2018), "How Satoshi Nakamoto and Bitcoin solved the double spending problem", available at: www.worldcryptoindex.com/how-bitcoin-solved-double-spending-problem/ (accessed 4 October 2020).

Yasin, R. (2018), "Protecting weapons systems against cyberattack: it's all about resilience", available at: https://federalnewsnetwork.com/cyber-exposure/2018/03/protecting-weapons-systems-against-cyber-attack-its-all-about-resilience/ (accessed 7 January 2020).

Author affiliations

Joe Garcia, Cyber Warfare Department, Naval Air Warfare Center Aircraft Division, Patuxent River, Maryland, USA

Russell Shannon, Mission Operations and Integration Department, Naval Air Warfare Center Aircraft Division Lakehurst, Lakehurst, New Jersey, USA

Aaron Jacobson, Cyber Warfare Department, Naval Air Warfare Center Aircraft Division, Patuxent River, Maryland, USA

William Mosca, Support Equipment Department, Naval Air Warfare Center Aircraft Division Lakehurst, Lakehurst, New Jersey, USA

Michael Burger, Mission Operations and Integration Department, Naval Air Warfare Center Aircraft Division Lakehurst, Lakehurst, New Jersey, USA, and

Roberto Maldonado, Cyber Warfare Department, Naval Air Warfare Center Aircraft Division, Patuxent River, Maryland, USA

Corresponding author

Russell Shannon can be contacted at: russell.shannon@navy.mil