




On business adoption and use of reproducible builds for open and closed source software

Simon Butler¹  · Jonas Gamalielsson¹ · Björn Lundell¹ · Christoffer Brax² · Anders Mattsson³ · Tomas Gustavsson⁴ · Jonas Feist⁵ · Bengt Kvarnström⁶ · Erik Lönroth⁷

Accepted: 10 October 2022 / Published online: 29 November 2022
© The Author(s) 2022, corrected publication 2024

Abstract

Reproducible builds (R-Bs) are software engineering practices that reliably create bit-for-bit identical binary executable files from specified source code. R-Bs are applied in some open source software (OSS) projects and distributions to allow verification that the distributed binary has been built from the released source code. The use of R-Bs has been advocated in software maintenance and R-Bs are applied in the development of some OSS security applications. Nonetheless, industry application of R-Bs appears limited, and we seek to understand whether awareness is low or if significant technical and business reasons prevent wider adoption. Through interviews with software practitioners and business managers, this study explores the utility of applying R-Bs in businesses in the primary and secondary software sectors and the business and technical reasons supporting their adoption. We find businesses use R-Bs in the safety-critical and security domains, and R-Bs are valuable for traceability and support collaborative software development. We also found that R-Bs are valued as engineering processes and are seen as a badge of software quality, but without a tangible value proposition. There are good engineering reasons to use R-Bs in industrial software development, and the principle of establishing correspondence between source code and binary offers opportunities for the development of further applications.

Keywords Reproducible builds · Software integrity · Software engineering · Open source software

1 Introduction

In his Turing Award Lecture in 1984, Ken Thompson described an attack on a computer system using a compromised compiler that injects malicious code into applications, and can also be engineered to cover its own tracks (Thompson, 1984). Known as “Trusting Trust”, the attack, if performed well, can be undetectable. At the core of the attack is the notion that users of software are required to trust that the software creator has delivered

✉ Simon Butler
simon.butler@his.se

Extended author information available on the last page of the article

an executable binary that operates only as claimed, and does not perform any hidden or malicious actions. Trusting trust is a problem that has fascinated some computer scientists and security experts, e.g. Wheeler (2005, 2009), who have tried to find solutions to the problem.

Most software is provided to users, in the wider sense, as software as a service (SaaS) solutions or in precompiled binaries. SaaS users rely on software running on remote systems over which they have little control, and limited information about exactly what software is deployed to provide the service (Tapas et al., 2019). In the case of precompiled binaries of operating systems and applications, most users download executable files and run them directly on their computers (de Carné de Carnavalet et al., 2014; Lamb & Zacchiroli, 2021). Users of proprietary software are required to trust the software supplier has distributed a binary that does not contain malicious code, despite there being extensive evidence that the integrity and quality of distributed software can be compromised (e.g. Edge (2019); Greenberg (2017, 2018); Smith (2011); GReAT AMR (2019); Gallagher and Greenwald (2014); Ohm et al. (2020); Ramakrishna (2021)). Open source software (OSS) can provide the opportunity for software users to verify the claimed correspondence between the source code and the distributed binary, because OSS projects create and distribute binaries of software built from specific revisions of source code which are also available. A reproducible build (R-B) allows the user of the software to build the binary independently of the software provider (Ren et al., 2018, 2019; de Carné de Carnavalet et al., 2014; Reproducible Builds Project, 2019a; Lamb & Zacchiroli, 2021). The user is then able to perform a bitwise comparison of the two binaries to verify that they are identical and that the distributed binary is indeed built from the source code in the way the provider claims. Applied in this manner, R-Bs function as a *canary*, a mechanism that indicates when something might be wrong, and offer an improvement in security over running unverified binaries on computer systems. The key property of a R-B is that it establishes correspondence between source code and binary. Potentially, such a simple property can be applied to support a wide range of software development activities including those that depend on source code audit, such as software quality assurance, supply chain integrity, and automation to support software licence compliance processes (van der Burg et al., 2014; Kuhn et al., 2020).

The use of R-Bs is gaining traction with some OSS distributions and projects. The Debian Linux project is a leader in the area with some 90–93% of packages built reproducibly (Ren et al., 2018; Levsen et al., 2019). Alpine Linux (Alpine Linux, 2020), Arch Linux (Vinet & Griffin, 2022), FreeBSD (Piotrowski, 2018), GNU Guix (2019), NixOS (2020) and the Yocto Project (2021) also provide R-Bs. Despite there being security, integrity (Lamb & Zacchiroli, 2021), and software quality benefits to R-Bs (Potvin & Levenberg, 2016; Bazel, 2020), adoption appears to have been slow and it remains unclear the extent to which R-Bs might become more widely used. Indeed, some Linux distributions have previously claimed they do not need to provide R-Bs because they have transparent and trustworthy development processes (Bressers, 2016). However, in doing so software suppliers continue to expect users to trust them without providing a means to verify that they are trustworthy, and sometimes with disastrous outcomes such as the attack on SolarWinds (Ramakrishna, 2021; Egts & Hellekson, 2021; Lamb & Zacchiroli, 2021).

Software engineering practitioners have suggested additional uses and benefits of R-Bs. Martin Fowler, for example, sees applications in the long-term maintenance of software to support debugging of earlier product releases installed at customer sites (Fowler, 2010). While others have also argued that R-Bs can lead to more efficient build processes as binaries are only recreated when there is a functional change, rather than as the consequence of a change in a volatile quality such as a file time stamp (de Carné de Carnavalet et al., 2014;

Ren et al., 2018, 2019). Further, some OSS projects where security is a very significant concern use R-Bs during the release process, where multiple developers build the software using a “recipe” that defines how to create a virtual machine, or container, as a clean environment within which to run the build process. The resulting binary is then digitally signed by each developer and the results compared (Bitcoin Project, 2022; Tor Project, 2022). Only when a consensus is achieved amongst developers that it is possible to build bitwise identical binaries independently is the binary released by the project.

The academic literature on R-Bs has largely focused on the technical challenges (de Carné de Carnavalet et al., 2014; Ren et al., 2018, 2019; Shi et al., 2021), on supply chain integrity in OSS (Lamb & Zacchiroli, 2021), and on specific implementations of R-Bs such as Guix (Courtès & Wurmus, 2015; Courtès, 2017), NixOS (Dolstra et al., 2010) and *in-toto* (Torres-Arias et al., 2019). The grey or practitioner literature¹ contains a range of views on R-Bs and some practitioners have identified additional applications of the techniques. There, however, has been limited research on the application and value of R-Bs in an industry context. This article explores the perceptions and applications of R-Bs in businesses in both the primary and secondary software sectors² in Sweden and Europe. The investigation is focused on the following three research objectives:

- O1: To understand the level of awareness of reproducible builds within software-intensive businesses.
- O2: To identify technical and business factors relevant to the use of reproducible builds.
- O3: To identify use cases for reproducible builds in a variety of technical domains and business contexts.

To meet the objectives we undertake two main phases of investigation, described in detail in Sect. 3. The first phase consists of discussions between the authors on the topic of R-Bs, and the relevance of the practice to their business. In particular we focus on the business, managerial and technical aspects of R-Bs relevant to the companies represented by the authors. The second phase of the investigation consists of analysis of interviews with practitioners about their understanding of the role the application of R-Bs might play in their work, and the opportunities and challenges they present both for themselves and for the software industry.

The following section describes current work on R-Bs and proposed solutions to the practical problems. Detail on the potential applications beyond trust identified by practitioners is also included, as are counter arguments of practitioners unconvinced by the approach or the need for it. Section 3 describes the research approach used in this work. Our findings are reported in Sect. 4 and discussed in Sect. 5. Finally, we summarise the contributions made by this paper in Sect. 6 before drawing conclusions and suggesting areas for future work.

¹ We use the term *practitioner literature* in this article and consider it to be synonymous with *grey literature*.

² Businesses in the primary software sector develop and sell software and software services. Businesses in the secondary software sector develop software components and incorporate them in other products (Ågerfalk et al., 2005).

2 Background and literature review

The motivation for the development of R-Bs has its roots in the need to verify correspondence between published source code and distributed binaries to provide an indication that additional code has not been inserted into the executable file during creation of the binaries (Reproducible Builds Project, 2019b; Porup, 2016; de Carné de Carnavalet et al., 2014). An example of the problem being addressed can be found in Thompson’s description of how a compiler might be compromised so that it inserts malicious code into a computing system in a way that is difficult to detect. Thompson summarises his experience and its implications as follows:

The moral is obvious. You can’t trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. (Thompson, 1984)

Given the volume and complexity of software that companies and individuals execute on computing devices today — e.g. cloud systems, desktop, gaming devices, mobile phones, and increasingly embedded systems in household devices — it is impossible for the user to create the code themselves. Consequently, computer users obtain much of their software from external sources either directly from the software provider or via a software distributor. A further complication is the pace at which revisions are made and programs updated on computers, which would require a similarly paced scrutiny process by software users. Largely, software users are forced into the position of trusting software distributors and providers, with only limited mechanisms to scrutinise the executable software (e.g. virus checkers and checksums), and with neither evidence nor methods to establish whether the software providers are trustworthy. Furthermore, software providers face the similar challenges to verify that their tool chains are trustworthy and reliable (Thompson, 1984; Xiao, 2015; Kang et al., 2015; Shaulov, 2016), particularly as supply chain attacks are seen as a relatively low cost mechanism for the distribution of malicious code, and are increasing in frequency (Ohm et al., 2020). A recent illustration is the large-scale supply chain attack on SolarWinds in 2020, which placed malicious payloads in versions of the Orion network management software that was subsequently downloaded by customers (Ramakrishna, 2021). The nature of the security breach and the relevance of R-Bs were summarised as:

...it appears that the **source code** wasn’t compromised, and the **distribution** system wasn’t compromised. Instead, the **build system** was compromised. This is **EXACTLY** the kind of attack that is countered by reproducible builds. Thus, the recent SolarWinds subversion is a very good argument for why it’s important to have reproducible builds (and to verify builds using reproducible builds).(David A. Wheeler in a message to the Reproducible Builds Project mailing list 2020-12-18³ (original emphasis).)

The Reproducible Builds Project defines an R-B as:

A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts. (Reproducible Builds Project, 2019a)

³ <https://lists.reproducible-builds.org/pipermail/rb-general/2020-December/002109.html>

2.1 Implementing reproducible builds

A key challenge for the software provider lies in creating a build environment and tooling that the user can replicate easily and independently, and that does not introduce nondeterministic artefacts into the binary. There are many sources of nondeterminism or variability in build environments that can be propagated to executable files, including time stamps and file paths, and variations in file sorting orders in different operating system locales that influence the order of compilation and linking and thereby the structure of the binary (Linderud, 2019). Indeed, de Carné de Carnavalet et al. (2014) argue that an important source of challenges for those trying to secure the supply chain is that the tools — compilers, linkers, build tools — have not been designed with the intention of creating reproducible output. The finding remains current and relevant according to practitioners interviewed by Enck and Williams (2022). There is then a twofold problem when trying to create a reproducible build: firstly the causes of nondeterminism in a build process must be identified, and secondly solutions that eliminate or control the causes of nondeterminism need to be found (Ren et al., 2018, 2019, 2022; Shi et al., 2021). Tools and techniques are being developed to support the identification of variance in build processes. For example, *reprotest* (Reproducible Builds 2022) can be used to build a software project using two different environments. The two binaries can then be examined with *diffoscope* (Reproducible Builds Project, 2022), a tool that performs a recursive `diff`⁴ on archive files, to identify possible sources of variability. Another approach traces system calls in the operating system during the build process to identify causes of variability or nondeterminism (Ren et al., 2019), which in turn can be leveraged to create patches to make software builds reproducible (Ren et al. 2022).

One source of variability in binaries has been the addition of variable timestamps, typically the time of compilation, by compilers and build tools. The Reproducible Builds project developed the `SOURCE_DATE_EPOCH` specification (Lamb & Luo, 2017) allowing a build time to be specified for compilation that is defined for the build and is independent of the system clock. A number of open source compilers and build tools, including GCC (2020), implement the standard. Additional methods are needed to control variability in some programming languages and build tools. Maven, for example, has recently implemented a plugin to make builds reproducible, but there are some constraints the user must adhere to for the build to be reproducible (Apache Maven, 2022). Another approach, adopted by Microsoft, is to replace timestamps with hashed values (Chen, 2018). However, not all programming languages and build tools have inbuilt support for reproducibility (Enck & Williams, 2022). A more recent approach has explored mechanisms to mitigate stochastic operating system behaviour to support deterministic execution (Navarro Leija et al., 2020).

Blaze (Ivanković et al., 2019) is a build tool developed by Google that implements R-Bs. It is also available as an OSS version named Bazel (2020). The motivation for Blaze and Bazel lies in the efficient use of computing resources. Google maintains a monolithic source code repository containing all its source code in which changes in one codebase trigger rebuilds in projects for which the project is a dependency (Potvin & Levenberg, 2016). Both Blaze and Bazel implement R-Bs for multiple programming languages so that the tools build only software components that have been changed in a meaningful way (Potvin & Levenberg, 2016).

⁴ `diff` is a command line program that outputs the differences between files.

Binary Authorization for Borg (BAB) is used by Google to demonstrate the integrity of applications uploaded by customers and deployed to their Borg cloud service to protect against tampering by staff (Google Cloud, 2020). Tapas et al. (2019) identify the need for and propose schemes to verify securely software deployed to users in SaaS systems. The proposed solutions rely on reproducibility, and on the use of *Merkle trees*⁵ to support verification of artefacts, including orchestration configuration and binaries (Tapas et al., 2019). Google have also developed systems to support build *verification* of libraries for the Go programming language, for example, that rely on Merkle trees to provide provenance (Hurst, 2021).

de Carné de Carnavalet et al. (2014) identify limitations to R-Bs, particularly with legacy OSS and closed source dependencies, and reason that a *verifiable build*, such as the approach described by Hurst (2021), where build differences can be accounted for, may be a necessary, pragmatic solution in some circumstances. Shi et al. (2021) developed a systematic approach to creating verifiable builds that was successfully applied in large-scale commercial software systems. The approach resulted in 100% of build artefacts in three large-scale systems at Huawei being verifiable (Shi et al., 2021).

2.2 Applications of reproducible builds

R-Bs have further applications in software development and deployment. OSS projects often provide both compiled binaries and the source code from which the binary is *claimed* to have been built, and, thus, may provide the opportunity to apply R-Bs. Some security software projects, including Bitcoin Core and the Tor browser, use R-Bs to support their release process by establishing that multiple, distributed developers are independently able to reproduce bitwise copies of the release candidate binary (Bitcoin Project, 2022; Tor Project, 2022; de Carné de Carnavalet et al., 2014). The intention is to eliminate a single point of failure, or point of attack, from the release process for secure software (Perry, 2013).

Fowler (2010) identified the value of R-Bs in software maintenance especially in continuous integration (CI) processes where software is released frequently. Fowler argues that being able to reproduce precisely the binaries deployed to the customer's site at some arbitrary point in the past is invaluable support for debugging and fault resolution. The NixOS (Dolstra et al., 2010) and Guix projects (Courtès, 2017, 2013) are Linux package managers designed to allow the user to reproduce software configurations. Both NixOS and Guix implement the functional specification of systems, where packages and their immutable dependencies are specified recursively, and a software build is considered to be a pure function, i.e. "a package's build function is assumed to always produce the same result" (Courtès, 2017). Courtès and Wurmus (2015) describe Guix-HPC (2020) a system for reproducing computational environments on computing clusters. Guix-HPC is designed to support the replication of computational experiments by the user in large multi-user systems, to support reproducible science (Courtès & Wurmus, 2015). However, it is important to note that there are further computational problems that may need to be resolved to make scientific findings reproducible (Wang et al., 2020). More widely, both Guix and NixOS, in combination with the Software Heritage project (Software Heritage, 2019; Rousseau et al., 2020), are also laying the foundation of reproducible computing environments to support long-term software maintenance (Courtès, 2019).

⁵ The cryptographic data structure underpinning blockchain.

Another approach, called DetTrace, implements a container for applications that controls sources of nondeterminism arising from the operating system at runtime (Navarro Leija et al., 2020). Consequently DetTrace allows software to execute deterministically and can, thus, build software reproducibly. Perhaps more importantly, DetTrace supports the deterministic execution of machine learning applications leading to repeatable and reproducible experiments in AI (Navarro Leija et al., 2020).

Recently, Bitcoin Core have started to use Guix to support their build process (Dong, 2019). That Guix is reproducible reduces the amount of trust the project needs to have in upstream systems and dependencies (Dong, 2019). Bitcoin Core has an acute concern about confidence in their reputation as software developers, which is related to security and to the quality of their software not least because Bitcoin transactions cannot be revoked. Given the value of the Bitcoin market there are significant incentives for malicious actors to compromise the software⁶. Dong is clear that R-Bs are one tool that supports a transparent and secure build process (Dong, 2019).

The challenges of managing third-party intellectual property (TPIP) in the software bill of materials (SBoM) (Riehle & Harutyunyan, 2019) and the SBoM of containers such as Docker (Hemel, 2020; Courtès, 2020), including licence compliance, can also be addressed by using R-Bs. Riehle and Harutyunyan (2019) outline the challenges of managing open source licence compliance in the SBoM where there is a mixture of proprietary and open source licensed components in a single product. A further problem is that there can be many versions of source code publicly available in multiple repositories and that there is therefore a broader concern of provenance in the long-term maintenance of software (Rousseau et al., 2020). The SBoM of a sample of seven OSS packages was investigated by van der Burg et al. (2014) who found a variety of inconsistencies and incompatibilities between the licences used in components. A challenge identified is that not all source code files distributed form part of the compiled software deliverable, thus the detection of licence compliance requires a detailed understanding of the build process used (van der Burg et al., 2014). Technologies such as SPDX (SPDX Workgroup, 2021) can be used in metadata to specify the licence or licences used by each dependency in the SBoM and the Open Source Tooling Group (OSTG) and Automated Compliance Tooling (ACT) (ACT, 2020) are developing solutions that use a combination of R-Bs and SPDX to automate licence compliance checks — licence clearance (Riehle & Harutyunyan, 2019) — in continuous integration (CI) through correspondence between the source code audited for licence compliance and the binaries created and integrated during the build process (Geyer-Blaumeiser, 2019). Furthermore, the use of R-Bs can contribute to securing CI pipelines by providing assurance that only audited SBoM components are integrated into the distributed software (Jacomet, 2020).

Courtès (2020) argues that containers, such as Docker, are often not created transparently, and that containers are not always reproducible. Variability in containers arises, for example, when the *recipe* or script building the container, or the startup script in the container itself, downloads packages from a software distributions' repository using a package manager such as `dnf` or `apt`. Consequently, a container specified and tested on one day can be a different binary, with potentially different and inconsistent behaviour, when deployed some time later following an update to any of the required packages in the distributions' repository introducing additional challenges for long-term software maintenance.

⁶ At the time of Perry et al. (2014) there was a market capitalisation of 4 billion USD for Bitcoin, and, at the time of writing, it is around two orders of magnitude greater. <https://www.blockchain.com/en/charts/market-cap>

A further concern is that containers are software distributions that result from compilation processes and it can be difficult to understand the SBoM deployed in containers, and accordingly the software licensing of containers has been highlighted by Hemel (2020) as legally problematic. Zerouali et al. (2019) also identify the security implications of containers where the user is uncertain what software has been built in to the container, or is downloaded and executed. Guix supports the creation of binary reproducible containers (Courtès, 2020). Another approach that secures the entire supply chain and the SBoM is taken by *in-toto* (Secure Systems Lab, 2022) which provides mechanisms to specify aspects of the software development process for components at each stage in the chain that can be verified in the following and subsequent steps using R-Bs (Torres-Arias et al., 2019).

2.3 Wider application of reproducible builds

Current applications of R-Bs are largely within a software development teams, both within companies and in distributed teams. The Reproducible Builds project consider how R-Bs might be used by a consumer such as an end user of software with limited technical knowledge, or someone who might lack the time to be able to rebuild distributed software. Individual OSS projects often provide both compiled binaries and the source code from which the binary is *claimed* to have been built, and thus provide the opportunity to apply R-Bs to verify the claim. Similarly, Linux distributions publish both binaries and source code. One anticipated mechanism is that a user might be offered an indication that m of n rebuilders confirm that a given binary is a reproducible before installing it (Nesbitt & Pounds, 2019; Levsen, 2016), i.e. that there is a degree of *consensus* amongst a group of rebuilders that the claim a given build is reproducible is correct. The proposition depends on there being a sufficiently large pool of rebuilders, in terms of numbers or quality, for the user to consider m of n to be meaningful in terms of trustworthiness. Chris Lamb of the Reproducible Builds project argues that a diversity of locations, legal jurisdictions, and computing systems is a desirable quality amongst rebuilders, so that the consensus reached on a particular build has the broadest possible provenance (Nesbitt & Pounds, 2019). What might motivate a diverse community of rebuilders is open to speculation. One reason may be a common need for security as illustrated by Perry et al. (2014)'s account of development of the Tor browser where a distributed group of rebuilders other than the developers add a layer of confidence to the claim that a given build is reproducible. The use cases for R-Bs identified in the academic and practitioner literature are summarised in Table 1.

The practitioner literature focuses on the challenges of implementing R-Bs, as well as identifying use cases. Some OSS projects, especially some security applications, Linux distributions and FreeBSD perceive threats to their reputations and have invested in developing R-Bs. Meanwhile, some OSS projects remain sitting on the fence or have been dismissive of the value of R-Bs. While technical aspects of the problem are being addressed and the value of R-Bs is clear to some practitioners and businesses, there is limited research literature that addresses how R-Bs are perceived within businesses that develop and deploy software.

Table 1 Use cases for reproducible builds reported in the academic and practitioner literature

Use Case	Description	References
Intellectual Property Management	R-Bs of audited source code to support licence clearance in complex SBOM Reproducible SBOM for containers	Geyer-Blumeiser (2019) Courtès (2020)
Reproducible Scientific Computing	Bitwise reproducible computational environments to replicate experiments.	Courtès and Würmus (2015)
Reproducible Systems	Reproducible Linux distributions	NixOS (2020), GNU Guix (2019)
Software Build Efficiency	Use of dependency resolution to rebuild only when necessary	Potvin and Levenberg (2016)
Software Maintenance	Bitwise replication of deployed software to support fault resolution.	Fowler (2010)
Supply Chain Security	Verification of R-B claim in single supply chain step	Reproducible Builds Project (2019b), de Carné de Carnaavalet et al. (2014), Ren et al. (2018), Perry (2013), Levsen (2016)
	Verification of all steps in supply chain	Torres-Arias et al. (2019)
	Development of consensus R-B amongst distributed developers, prior to release	Perry (2013), Dong (2019)
	Secure deployment of customer software in cloud computing systems.	Google Cloud (2020)

3 Research approach

In this investigation, we adopt an action-case approach (Braa & Vidgen, 1999; Lundell & Gamalielsson, 2017) to explore the uptake and possible use cases of R-Bs in primary and secondary software sector businesses. The research was conducted in two main phases. The initial phase consisted of discussions between the co-authors with the purpose of understanding how reproducible builds are and might be used within the companies represented by six of the co-authors, as well as seeking to understand the business cases for adopting and using R-Bs. The second phase of the research consisted of interviews with practitioners from businesses and organisations developing and applying R-Bs.

We used our networks to identify individuals, both decision-makers and practitioners, who had experience of R-Bs in their work, or who had considered, or were in the process of considering, using R-Bs. Where possible we were introduced to potential interviewees through the networks. We also identified other individuals active in R-Bs communities or advocates for R-Bs within OSS projects, and where we were unable to contact them through our networks, we made a direct approach by email, inviting them to participate in an interview.

Interviews were conducted as open dialogues, in English by the first author, a native English speaker, by email, online conferencing, or telephone according to the interviewee's preference. The interviews explored the interviewees' knowledge and experiences of R-Bs, and the business and technical grounds for decisions they had made, and the business and technical use cases for R-Bs. The interviewer used a list of questions to guide the conduct of each interview (see Appendix B). Only one interview was conducted by email. The interviewee was very expansive, so required little prompting to discuss their work and ideas.

Interviews were transcribed by the first author, and the transcript anonymised. Each anonymised transcript was reviewed and approved by the interviewee. Interviewees were also invited to expand on points made during the interview. A brief synopsis was also agreed characterising the interviewee's role and business context in abstract terms (see Table 3) as well as a synopsis of the interview including quotes, identified by the first author, that might be used in publications. The intention of asking the interviewees to review the anonymised transcripts and synopses was to take a step towards ensuring that the transcription accurately reflected the interviewees' views, and to reduce researcher bias.

There was one exception to the process described. One interviewee is responsible for aspects of their employer's software security and was interviewed by managers within the business to ensure sensitive information was not disclosed during the interview. The first author was given written notes of the interview and was able to discuss the interview with the managers who had conducted it. Both managers are software practitioners with extensive experience (in excess of 20 years) and are familiar with the interviewee's work. The interviewee was able to respond to follow-up questions submitted via the interviewers.

Interviewees were all experienced software practitioners with a minimum of five years industrial experience. Companies employing interviewees — the six represented by the authors, and six others based in Europe — ranged in size from small (<5 employees) to companies with multiple divisions and thousands of employees.

The anonymised interview transcripts were analysed by the first author using thematic analysis (Braun & Clarke, 2006). The academic and practitioner literature explored in the preceding section, and the investigation undertaken amongst the authors reported in Sect. 4.1 was used to develop semantic themes before the interviews were conducted. Themes on the applications of R-Bs and the challenges of implementation (see Table 2),

Table 2 Semantic themes related to the applications and implementation of R-Bs derived from academic and practitioner literature, and investigation for OI

Grouping	Theme	Description
Application	Code-audit-dependency	The use of R-Bs to support dependency resolution.
	Code-audit-integrity	The use of R-Bs to establish a connection between source and binary.
	Code-audit-legal	Use of R-Bs as evidence of the source code used in a particular binary.
	Code-audit-safety	The use of R-B to audit code deployed in safety-critical environments/contexts.
	Long-term-maintenance	Application of R-Bs to support software maintenance in the longer term (>10 years).
	Reproducible-science	The use of R-Bs to support reproducibility of scientific results.
	Scripting-languages	The application of R-Bs to the distribution of libraries implemented in scripting languages.
	Software-development-traceability	The use of traceability in software engineering processes, regardless of the use of R-Bs.
	Challenge-external	That implementation of R-Bs is constrained or prevented by factors outside the business's control.
	Challenge-internal	That existing company internal processes are sufficient, or introduce challenges that cannot easily be resolved.
Implementation	Challenge-technical	That the implementation of R-Bs arises from technical sources including tooling.

Table 3 Interviewee roles and technical domains in which they operate

Identifier	Role		Domain
	Managerial	Technical	
<i>I</i> ₀₁	✓	✓	Embedded Systems
<i>I</i> ₀₂		✓	Safety-Critical Systems
<i>I</i> ₀₃	✓		Research
<i>I</i> ₀₄	✓	✓	Research
<i>I</i> ₀₅	✓	✓	Web Systems
<i>I</i> ₀₆	✓	✓	Embedded Systems
<i>I</i> ₀₇	✓	✓	Safety-Critical Systems
<i>I</i> ₀₈	✓	✓	Safety-Critical Systems

and the motivations for their use — including business, security and technical motivations (see Table 6 in Appendix A). The themes were used to support the theoretic thematic analysis of the interviews, thereby facilitating the identification of novel applications of R-Bs or motivations for their use. Synopses of the anonymised interview transcripts and the thematic analysis were discussed by the authors to develop the findings and analysis (McDonald et al., 2019). The evolved results were also subject to additional scrutiny in an iterative process involving researchers and practitioners from the businesses represented by the authors during in-company workshops and as part of four full-day workshops (Lundell & Gamalielsson, 2017).

4 Findings

4.1 O1: Business awareness of R-Bs

The six businesses represented by the authors are based in Sweden and operate in the primary and secondary software sectors developing solutions mostly in the high performance computing (HPC), internet of things (IoT), safety-critical systems, and security domains. As a group of businesses we, with one exception, do not currently use R-Bs in our work. Awareness of R-Bs was mixed within the group of authors at the beginning of this research, with the greater knowledge amongst those working in the safety-critical and security domains.

We identified three areas in which R-Bs are or may be of value as day-to-day software engineering practices within the six businesses. The first is the verification of software binaries distributed by OSS projects. Much of the OSS used in systems we develop is built from source, in some cases we are building on the software before contributing revisions upstream, or there is a need to audit the source code for reasons including licensing and security. The second is the practical value of R-Bs in software development, particularly when working with complex and safety-critical systems, R-Bs can contribute to certification processes and reduce the need for code audits. The third is the value of a verifiable process where we distribute OSS. This is not just in the first sense, but potentially offers business value as a demonstration of the integrity of internal software processes in the development of security applications.

Importantly, the fundamental property of a R-B — correspondence between source code and binary — has a wide range of potential applications. We encounter areas of work, particularly in safety-critical and complex, multi-component systems, where traceability and software provenance are useful tools to ensure that only audited software is deployed and that systems consisting of versions of software known to work together are assembled in tested configurations and combinations. An example is a train that may contain 30 to 40 interconnected computers, some of which are components of the safety-critical functions of the train, whereas other components are more related to the business and social aspects of managing a train, such as ticketing systems and CCTV. One of us (Brax) was able to identify more than 200 applications running on one train during a recent assignment.

Component safety-critical systems are typically tested in a laboratory setting using a system that simulates the inputs to the safety-critical system under test so that the software can be exercised to ensure that it meets or continues to meet the certification requirements. Of considerable concern is the impact on this process of managing configurations of multiple computing devices each with its own combinations of hardware, firmware, operating system, libraries and applications, all of which will change over time in each device. Currently, many testing systems are configured manually, but there are strong arguments for using automation and being able to reproduce deployed software reliably through the application of R-Bs is a potential solution to the problem.

The development of aviation software of software for flight safety-critical applications is governed by the RTCADO-178C guidelines (RTCA, 2011). RTCADO-178C:

provides the aviation community with guidance for determining, in a consistent manner and with an acceptable level of confidence, that the software aspects of airborne systems and equipment comply with airworthiness requirements. (RTCA, 2011)

Compliant software processes are developed by companies and certified on a per-project basis (Pothon & Ochem, 2017). Correspondence between source code and binary is a fundamental requirement of DO-178C which extends to reproducible test environments to support the software maintenance. Given audited and tested source code that meets the standard, then a R-B might be used to meet the requirement that accurate copies of certified binaries can be recreated (RTCA, 2011, Chapter 7.2.7.c). To fulfil the requirements of DO-178C not only product binaries need to be saved, but also the hardware, software tools and build environment used during the development needs to be documented, saved or stored.

The aeronautics division at Saab use R-Bs to meet the requirements of RTCADO-178C. One requirement is that it must be possible to build a new binary again and trust all previous verification, test and statements made in the documentation (RTCA, 2011, Chapter 8.3.f). Metadata is stored identifying the versions of source code used in builds, as well as records of checksums calculated for the binary files created. The reproducible build process then creates bitwise identical binary copies of previous builds, that correspond to audited source code. Saab also use R-Bs to support some of their software development processes in other divisions where the adoption and use of R-Bs as best practice has been led by software practitioners.

Commonly used arguments for R-Bs to be implemented as part of the process for securing the supply chain in OSS apply to PrimeKey who offer services and software in the security domain focused on public key infrastructure (PKI). PrimeKey lead a number of OSS projects and see business value in providing a verifiable correspondence between delivered binaries and audited, secure source code for some customers. As with any OSS project, vulnerabilities are not just restricted to the build tool chains, but also the developer accounts. Consequently an ideal process would consist of source code audit

which is then built using an automated R-B. Currently, customers are not asking for R-Bs as part of the software development process, but R-Bs can be seen as a way of adding value for customers with a high-level business process perspective of security. PrimeKey have experimented with R-Bs — an initiative led by developers — but found support for R-Bs in Maven⁷ was insufficiently mature at the time.

In summary, amongst six businesses in the primary and secondary software sectors there is experience of using R-Bs in production in one company, and an understanding of potential applications and business value in two more. In the case of PrimeKey tooling was an obstacle to implementing R-Bs.

4.2 O2: Relevant technical and business factors

Interviewees reported a mixture of motivations to use R-Bs, as well as challenges or obstacles to their implementation. The majority of motivations stated were technical, and some interviewees, including I_{01} see the value of R-Bs to their workflow, both as a way of being able to verify the correspondence between binaries and source code from suppliers and as a mechanism to improve traceability of the code they create, package and deploy to customers. Considering the use of R-Bs from the perspective of a consumer of software, I_{01} summarised that matter succinctly:

...the guarantee of reproducible builds would make me feel safer when I get software from somebody else.

I_{01} also summarised the application of R-Bs in the relationship with customers from two perspectives: practical and legal. The traceability facilitated by R-Bs supports software maintenance:

...for me it is quite important to be able to know what actually happened and trace back to the source code. I_{01}

And, further, that as well as being a source of evidence to support engineering, I_{01} made the point that:

...reproducible builds is very important in forensics - when you have legal issues about code ...

I_{08} described using Debian Linux as the starting point for a secure OSS supply chain within a business that develops safety-critical software. Debian was chosen because of the implementation of R-Bs and the provision of metadata to support the process. The business is then able to rebuild the distribution for itself, package by package, establishing trust in the supply chain upstream. As part of the process of creating the internal distribution the business undertakes additional code audit steps using static analysis, as well as performing licence clearance — checking for licence statements in source code and other artefacts — and looking for export-controlled technologies. The code audit establishes the provenance, licensing and security of the SBoM in the Linux distribution, and allows the business to deploy a trusted platform with a known SBoM for internal development. While this method might seem costly, and is perhaps only possible — at the level of a Linux distribution — for a large company, I_{08} identified additional engineering and business benefits that we will return to below.

⁷ <https://maven.apache.org/>

Both I_{07} and I_{08} identified the challenge of developing trust in upstream developers. For I_{07} a problem was that OSS projects make revisions to functionality that are acceptable to the project. Software development teams taking OSS into a secure or safety-critical environment then have to audit and review revisions to source code in bug fixes and new features before being able to integrate the code into their products, according to I_{07} . A concern I_{08} expressed was the difficulty of estimating when to start evaluating revisions to source code to allow for system upgrades in the upstream distribution to be incorporated in the local Linux distribution. A task made more demanding by working with complex dependency trees in both the current version of the upstream distribution and the development branch that will become the next revision. I_{08} also reported that a key technical benefit of developers using a mirrored, trusted Linux distribution is that there is better control of internal software builds in product development, which leads to fewer problems for both developers and end users, and reduced maintenance costs.

Interviewees also identified obstacles to using R-Bs. In the context of bitwise reproducibility, I_{02} stated that:

...it would be great to have reproducibility for us, but not mandatory for us in our context. I would not consider that ...it is more important to have traceability.

I_{02} went on to say that it would be "...too much of a burden for us to move to reproducible builds." As well as the effort of implementing R-Bs and retrofitting them to existing build systems and technical obstacles to implementing R-Bs in some programming languages (see Sect. 2), some interviewees expressed concerns that the value to customers was limited, and thus there was a limited business case for using R-Bs. Indeed I_{04} argued that:

...the use of reproducible builds within a proprietary software development environment may be good from an engineering standpoint for developers in that company, but it has no tangible impact on users of the software.

I_{03} was similarly sceptical, arguing that some offerings based on reproducible builds may work with proprietary software, but generally the threat model is so diverse and at so many different levels — hardware, firmware, operating system, etc. — that applications of R-Bs are not obvious.

There is, as I_{04} observed, a difference between business models used by proprietary software and open source software. I_{04} went on to say:

Similarly, for free software companies that publish container images of their software (Docker, Flatpak, etc.), I would argue that providing provenance information that can be used to rebuild the images should be a good commercial argument.

I_{04} also expressed the opinion that R-Bs would become an integral part of conventional software development practice, saying, "I think reproducible builds will become the norm for developers, just like version control." The concern for many (similar to the views expressed in Sect. 4.1) is that the value proposition — how the benefits of a product are perceived by the customer — is not clear in financial terms and is more intangible. I_{05} , for example, argued that R-Bs might be seen as a mark of quality, implying that a software provider used good engineering practices. Similarly, I_{06} argued that reproducibility is a key characteristic of professionalism in the embedded systems domain.

A key challenge for R-Bs also identified by interviewees is that of awareness. Not just that businesses buying software and services may not be aware of the value of R-Bs, as I_{01} argued, but I_{05} also suggested that some practitioners may know about R-Bs and understand it as a good engineering practice, without seeing a business application.

4.3 O3: Use cases for R-Bs

In the latter part of the background (Sect. 2) we summarised the use cases for R-Bs found in the practitioner literature (see Table 1). The interviews explored the known use cases and their application in businesses, uses of R-Bs developed by businesses, and possible future use of R-Bs (see Table 5).

I_{03} argues that a key functionality of R-Bs is that of *dependency resolution*, and that it is clearly achievable given the example of Guix-HPC. I_{03} further stated that software licence audits are supported by dependency resolution, and that support for dependency resolution in docker containers is desirable.

The complexity of security in cloud applications was also highlighted by I_{05} . As noted in Sect. 2, Google use reproducible builds to guard against tampering with customer applications. Cloud providers, according to I_{05} , are also concerned about the provenance of the binaries they use, and consistency of deployed platforms. I_{04} considers the security applications of R-Bs as one that will attract potential users.

Where R-Bs were applied to establish supply chain security, interviewees also reported that R-Bs were often also applied to support the integrity of systems deployed to customers and traceability. I_{02} has considerable experience in delivering reproducibility to support long-term maintenance, highlighting the need to take snapshots of their builds to support long-running projects, often implemented and executing on legacy systems:

...we can freeze the context ...it is important for our customers in [the] aerospace industry where they have running projects for ten years or twenty years.

The company has developed a non-R-B solution that I_{02} describes as meeting their requirements:

...we can reproduce a build completely from any time in the past and that is important for us ...for that we add traceability information in our binaries that are generated so we track exactly what has been used to do a build.

I_{07} works in a similar safety-critical area to I_{02} and also discussed the application of R-Bs in long-term maintenance. Each internal release of deployed software needs to be maintained for thirty years, and I_{07} framed the problem, similarly to I_{01} and I_{02} , in terms of support for traceability. Teams managed by I_{07} introduced R-Bs as part of a series of gradual changes intended to improve to the business's software development process. I_{07} attributed some of the benefits observed directly to the application of R-Bs including an increase in the efficiency of build processes. A reason given by I_{07} was that increased scrutiny of code during the process of modularisation resolved a significant amount of technical debt. The teams also found that dependency management was simplified by the use of R-Bs and I_{07} highlighted that long-term maintenance is much easier to support.

In Sect. 4.2 we reported I_{08} 's work to rebuild Debian reproducibly to establish a trusted Linux distribution for use within the business. An additional advantage I_{08} identified concerns collaboration with external partners. I_{08} 's organisation can exchange project source code, specify the build system, and supply a checksum for the binary to external companies. Knowing that their internal Linux is built reproducibly from the upstream distribution, they can be confident that external partners can easily, and inexpensively, replicate the build system and environment for the collaborative project. Consequently, partners in any collaboration can, firstly, build software projects to create bit-for-bit identical executables and, secondly, understand if there is a difference in the build platforms should the R-B fail.

Another interviewee, I_{01} , identified R-Bs as a mechanism for establishing the integrity of deployed systems. Currently, I_{01} implements system integrity checks in embedded systems at boot time, so that incompatible configurations of hardware and software fail safely. I_{01} sees that R-Bs can be used to implement a more secure mechanism for ensuring that a fully tested software configuration is deployed, and as a means of being able to reproduce, debug, and deploy debugged systems to customers should problems arise. Traceability through R-Bs in this context allows the system developer to be certain what software and hardware were deployed in the event of any legal claim involving the system, and thereby its creator.

Considering the breadth of purposes to which R-Bs may be applied, I_{03} made the point that R-Bs are challenging to implement and binary reproducible builds are unlikely to be universally guaranteed. They can, however, be essential in specific, critical applications, including bitcoin, for example.

When asked about the future for R-Bs, I_{05} drew a parallel with the early development of distributed version control systems (DVCS), pointing out that DVCS was initially created to meet specific use cases and the way in which the technology is applied now could not have easily been conceived of at the time. There were obvious business and engineering reasons to use version control, but the reasons to the switch to a distributed version control system were less clear. Similarly with R-Bs, there are some, like I_{04} , who think that R-Bs will become a conventional software engineering practice, but, like I_{05} , expect that some future applications of the technique will be surprising.

4.4 Summary of findings

In the introduction to this paper we identified three objectives for the research that focus on business awareness of R-Bs, the technical and business factors that affect adoption and use, and the use cases. We briefly summarise our findings for each objective.

4.4.1 O1: Business awareness of R-Bs

Within the six companies represented by the authors we found awareness of R-Bs greatest within businesses working in the safety-critical and security domains. Within those companies the application of R-Bs has been largely instigated and led by engineering staff, and applied in specific workflows. Most notably to support the certification of a software development process used in avionics. Awareness extends beyond the companies applying R-Bs and two other companies have either experimented with using R-Bs or are considering future application of R-Bs in their work. We conclude from this small sample, that there is awareness amongst businesses of R-Bs as a software engineering technique, and that it is mostly the technical and engineering staff, particularly in companies in the security and safety-critical domains, that have knowledge of R-Bs.

4.4.2 O2: Relevant technical and business factors

Interviewees perceived R-Bs as good engineering practice that are of value to the business applying them (see Table 4). Apart from R-Bs being good engineering practice and perhaps having some value as a badge of quality, there appears to be limited business motivation in terms of adding value to the product that a customer might pay for. The motivation to use R-Bs arises from engineering concerns, as a means of supporting software development

Table 4 Summary of relevant business and technical factors identified by interviewees

Factor	Description	Interviewee(s)
Business Factors		
Value	<p>Business value Application of R-Bs has a value for the business.</p> <p>Customer value The application of R-Bs has a value (benefit) for the customer.</p> <p>Engineering value That the application of R-Bs has a value as an engineering process.</p> <p>Business awareness That practitioners within a company may not perceive R-Bs as something with business value.</p>	<p>I_{01}, I_{04}, I_{06}</p> <p>I_{05}, I_{08}</p> <p>$I_{03}, I_{04}, I_{05}, I_{07}, I_{08}$</p> <p>$I_{05}$</p>
Awareness	That practitioners within a company may not perceive R-Bs as something with business value.	I_{05}
Cost	<p>Customer awareness That customers may not understand what an R-B is nor why it might used.</p> <p>Cost of implementation Retrofitting R-Bs to existing build system is costly</p>	<p>I_{01}</p> <p>I_{02}</p>
Technical Factors		
Build tools	<p>Build efficiency The application of R-Bs improves the efficiency of the software build process.</p> <p>Programming language & build system limitations Not all programming languages and build systems support R-Bs</p>	<p>I_{07}, I_{08}</p> <p>I_{02}</p>
Security	<p>Distributed development R-Bs can be applied as part of a secure development or deployment process.</p> <p>Deployment R-Bs can be used to support a secure deployment process.</p> <p>Supply chain Provide assurance for software taken into a business.</p>	<p>I_{01}</p> <p>I_{01}</p> <p>I_{01}, I_{07}, I_{08}</p>
Traceability	<p>Generic R-Bs can be used to implement traceability</p> <p>Deployed configurations R-Bs can be applied to support software maintenance.</p> <p>Evidence R-Bs as a source of evidence in legal disputes</p> <p>Licence clearance R-Bs support linking of audited code</p>	<p>I_{01}, I_{02}</p> <p>$I_{01}, I_{03}, I_{04}, I_{07}$</p> <p>$I_{01}$</p> <p>$I_{03}, I_{07}, I_{08}$</p>

where assurances are required that there is a deterministic relationship between the source code and the binary. Return on investment for businesses appears to be a consequence of increased efficiency for developers, improvements in engineering quality, and potential advantages in software maintenance. Amongst the interviewees there was also an awareness of the challenges of implementing R-Bs, including cost, that can inhibit adoption, particularly where an existing process achieves related outcomes and gains from adopting R-Bs would be small.

4.4.3 O3: Use cases for R-Bs

A R-B has the underlying property of correspondence between the source code and the binary. Consequently there are likely to be a wide range of possible use cases that can rely on such a simple and generic principle; indeed one interviewee reasoned it is impossible to foresee the potential applications of R-Bs. From the academic and practitioner literature we identified the use cases given in Table 1, and interviewees discussed many of the same use cases (see Table 5). Through the interviews reported and within the companies represented by the authors, we identified the following use cases in addition to those given in Table 1 and Table 5:

- Certification of development process — the application of reproducible builds within a software development project to meet the requirements of a safety-critical, or other, standard or certification body.
- Collaboration — having established trust in a Linux distribution through rebuilding it reproducibly allows a business to collaborate confidently with partners using the publicly available distribution of the platform at no additional cost to collaborators.

In addition, the use of R-Bs to establish the integrity of a deployed software configuration at runtime was described by an interviewee and considered by one of the authors. We have not described this as a separate use case, rather seeing it as an extension of existing approaches to use R-Bs to implement traceability during software development, for example for licence clearance.

5 Discussion

The parallel drawn by one interviewee between the applications and uptake of distributed version control systems (DVCS) and that of reproducible builds is helpful from both technical and business perspectives. The sense in which the observation was made concerned the difficulty of predicting future technical applications of R-Bs. There may also be similarities from a business perspective: DVCS is not something that adds value to a product, and neither does it seem from interviewees' responses that R-Bs will add value directly. As with DVCS, it appears that R-Bs are an engineering process that improves the software development process, and have an economic benefit through efficiency gains, as well as improvements in software quality and integrity. However, as some interviewees noted, the cost of implementing R-Bs may be too great, but in some domains R-Bs may become standard practice given the need for traceability.

Table 5 Use cases for R-Bs discussed by interviewees

Use Case	Description	Interviewee(s)
Intellectual Property Management	R-Bs of audited source code to support licence clearance in complex SBoM	I_{07} , I_{08}
Software Build Efficiency	Reproducible SBoM for containers	I_{04}
Software Maintenance	Use of dependency resolution to rebuild only when necessary	I_{07}
	Bitwise replication of deployed software to support fault resolution.	I_{01} , I_{02}
Supply Chain Security	R-B used to support deployed system integrity check	I_{01}
	Verification of R-B claim in single supply chain step	I_{02} , I_{03} , I_{04} , I_{07} , I_{08}
	Development of consensus R-B amongst distributed developers, prior to release	I_{03}
	Secure deployment of customer software in cloud computing systems.	I_{04} , I_{05}
Collaboration	Use of reproducibly built Linux distribution to facilitate R-Bs in collaborative software development	I_{08}

The process of rebuilding a Linux distribution to establish trust in the supply chain as described by I_{08} requires an investment of resources that are likely only to be available to larger businesses. However, it brings benefits not only to the company, but also to those companies that work with it as collaborators or subcontractors. Having verified the Debian builds for itself, the company is able to establish that its collaborators and subcontractors are using the same distribution when they are able to reproduce binaries of collaborative projects bit-for-bit. There is reciprocity because collaborators are equally able to confirm that the company is using the trusted distribution and build tools it claims to, so, despite the investment being made by the larger business, the relationship may be more symmetrical than first appears. Indeed, it might be argued that a reproducibly built Linux distribution, such as Debian, where there are sufficient trusted rebuilders, would allow developers and companies to collaborate and establish trust in the collaboration by being able to build bitwise identical binaries from the source code being collaborated on.

Interviewees also discussed long-term maintenance scenarios where software has a working lifespan of decades. In one case, though not using bitwise reproducible builds, the need was to support a range of systems so that each software build was documented and reproducible. In another case, and also the certified development process described in Sect. 4.1, R-Bs are already being used in industry to support long-term maintenance. We speculate that the availability of distributions such as Guix and NixOS that apply reproducibility to support the deployment of reproducible systems, as well as Debian will lead to the use of R-Bs in long-term software maintenance becoming commonplace. When coupled with the distribution of trust in reproducible distributions, there may be further opportunities for long-term software maintenance, perhaps, for example, being able to accurately and confidently recreate legacy software and subsequently maintain it, for example, to maintain long-lived engineering artefacts, or read archived documentation.

As noted earlier (Sect. 2), software supply chain attacks have led to practitioners identifying build environment security, software auditability and reproducibility as areas in need of urgent attention (Enck and Williams 2022). There is interest in industry in reproducibility for software security and provenance (e.g. Chen (2018); Shi et al. (2021); Hurst (2021)), but the emphasis is on the software development process. The value, or potential value, to end users of software outside the development process is articulated less often. I_{04} argues that there is good commercial reason for vendors to provide provenance information for container or other images of their software so that they can be rebuilt. We would argue that the need for software provenance and transparency is great and that the use of R-Bs and other mechanisms to support provenance is becoming imperative. Tapas et al. (2019) identify the need for transparency and provenance in SaaS systems so that users are aware of what software is deployed for them to use. The authors identify security and privacy threats amongst their motivating examples, and we would add the reliability of the software delivered via SaaS, particularly where the software user has legal obligations, such as accounting software where accuracy and consistency are required⁸, or privacy obligations such as those under the GDPR (European Council 2016). Certainly there is a need for provenance systems for SaaS to include all dependencies of the running system. Regulatory obligations for reproducibility and transparency in software can further develop as the use of AI increases. There are strong arguments advanced in support of transparent and interpretable AI systems (Rudin 2019) and the development of legislation and regulations

⁸ The Horizon scandal in the UK provides an illustration of an unscrupulous software supplier (Peachey, 2022).

for AI in the US (Johnson 2020) and the European Union (European Commission 2021) is starting to support interpretable AI. The use of R-Bs in aircraft certification we give in Sect. 4 illustrates the potential for the application of R-Bs in software provenance for AI systems, though further mechanisms would need to be developed to support reproducibility of training sets, machine learning models, and system behaviour.

A challenge during this research has been the limited theoretical modelling of R-Bs which has made reasoning about R-Bs less than straightforward, and may be an obstacle to the identification and development of further applications. As a work uncovering practical applications and industry attitudes, such a task is beyond the scope of this article. However, modelling R-Bs, formally or semi-formally, is a topic for future work. For example, such modelling could support a better understanding of the costs and benefits of collaborative use case described in the preceding section. While it is clear that there is a, perhaps unexpected, benefit to verifying that a Linux distribution is reproducible, two questions arise. The first is: whether the benefits to the company investing time and effort on verifying the Linux distribution is reproducible outweigh the costs? The second question is: whether there is a pattern in this use case that can be applied in other situations? A formal or semi-formal model could help support such reasoning, and perhaps support the discovery of further applications of R-Bs.

From the use cases reported in the practitioner literature and uncovered during this research, some relevant dimensions or elements of a model might be inferred. One dimension of a model might reflect, at an abstract level, *why* the R-B is being used. R-Bs are used to ‘capture’ source code state in a code audit. The purpose the R-B serves might be to ensure only audited code is used in a software build to support licence clearance, or that the R-B is used to establish trust in the software supply chain with users.

Another dimension to consider is *who* uses the R-B. In the use case illustrated by Debian Linux the user of the R-B is the user of the distribution. In long-term maintenance and software traceability scenarios, for example, the consumer of the R-B is the software development team, as it is in the cases of the Tor browser and Bitcoin Core during their initial build. The latter instances illustrate that *when* may also be a consideration for any model; both in terms of a milestone and the time available before that point in time. Tor and Bitcoin Core, for example, use a distributed process to establish a consensus amongst developers that the build is reproducible *before* the software is released. After release the R-B can then be used to build trust by establishing that a user can create a bitwise copy of the binary from the source code. In these cases and that of Debian Linux there is, consequently, more than one group for whom the R-B has significance at different times. Furthermore, a distributed software development process, such as that used in Tor or Bitcoin Core, the initial time frame for rebuilding, so that the result is relevant for the developers, will be shorter in comparison to that for verifying a distribution.

An additional dimension for any model appears to be the *number of rebuilders* required for an R-B to be considered to have achieved a desired outcome. In some cases, such as long-term software maintenance, it may be sufficient that the R-B is successful, i.e. that the software can be rebuilt reproducibly. In such cases, the notion of independent rebuilding used to support trust in the software supply chain is less important, though of course any rebuilder in the future will, of necessity, be independent of the original builder.

Threats to validity With any empirical study there are threats to validity. We consider threats to construct validity and external validity. Threats to internal validity are not considered because no claims for causality are made, and statistical conclusion validity is not discussed because no statistical inference is used. There is a threat to *construct validity* from the initial semantic thematic analysis of the interviews being performed by a single author. The threat is mitigated in two ways. Firstly, the thematic analysis was informed by

the academic literature and, secondly, iterative discussions between the authors of interview summaries and analysis were used to refine the thematic analysis. Further the study and results have been subject to wider scrutiny at workshops involving practitioners from the companies represented by the authors.

Threats to *external validity* arise from the small number of interviewees and that the authors and interviewees are largely based in Europe. The threat to generalisation is mitigated by the diversity of size of the software-intensive businesses represented by the authors and the interviewees, as well as the types of industry within which they operate, and the domains in which they develop software. Although the companies are based in Europe, the companies, interviewees and authors also work in other jurisdictions with operational offices, collaborators, and partners in other countries and on other continents. Accordingly the perspectives reported can reflect current industry use of R-Bs in a wider context. Further, the types of decision reported by interviewees concerning the technical and business motivation to use R-Bs as well as the obstacles encountered, and the diversity of applications appear to be relevant to software-intensive businesses. Future work might extend the study by increasing the sample size; perhaps by conducting an online survey, for example. Such a survey could provide a broader picture of the use of R-Bs in industry, and could also serve as a means of identifying further interviewees.

6 Conclusions

In this article we have reported the findings of a study of the perception and use of reproducible builds (R-Bs) in businesses in both the primary and secondary software sectors. Most existing research on R-Bs is technical in nature and describes the development or application of R-Bs in specific contexts. The paper's chief contribution is to provide a picture of the developing use of R-Bs in industry including the use of R-Bs to support certification processes and collaboration between businesses. We also report a range of opinions reflecting the value of R-Bs to businesses and active domains of application, as well as the commercial challenges and advantages of using reproducible builds.

This study makes the following contributions:

- Identification of novel applications of R-Bs used in industry;
- Evidence that businesses understand the value R-Bs contribute to their software engineering and software quality processes; and
- That businesses mostly perceive R-Bs to be an intangible value proposition.

Reproducible builds are an engineering approach that have a simple principle at the core and consequently appear to have many possible applications. Our study has shown that the use of R-Bs is wider and more innovative than previously documented. Further, we found that R-Bs may be applied to reduce software development costs for some businesses, and provide opportunities for some ways of working. We expect increasing adoption of R-Bs as the techniques and applications become more widely known, and the benefits to engineering processes become better understood. To that end we suggest that future research in the area of R-Bs might explore the development of models and possible uses of the techniques to support greater understanding and reasoning about potential areas of application. As one interviewee observed, experience shows that we cannot predict how a particular technology may develop and might be applied; accordingly we look forward to seeing further development of industrial applications of reproducible builds.

Appendix A. Motivations for the use of reproducible builds

Table 6 Semantic themes capturing motivations for the use of R-Bs derived from academic and practitioner literature, and investigation for OI

Grouping	Theme	Description
BusinessMotivations	Engineering	That a business perceives a direct engineering benefit to using R-Bs.
	Reputation	A business sees the use of R-Bs as something that is reputational, e.g. the application of R-Bs is a mark of integrity.
	Value-buying	That the use of R-Bs adds value to a product being bought.
	Value-selling	That the use of R-Bs adds value to a product being sold.
SecurityMotivations	Anomaly detection	R-Bs to detect anomalies in software builds.
	Extent-of-impact	R-B use motivated by possible extent of security compromise, e.g. Bitcoin.
	Reputation	Motivation arises from perceived consequences of attack to individual or company (organisational) reputation.
TechnicalMotivations	Scope-vector-developer	R-B motivated by security threat to developer, or related to the potential of developer compromise.
	Scope-vector-software	R-B motivated in terms of the security threat to the software, e.g. supply chain security.
	Trust	Motivation for R-B in terms of establishment of trust.
	Build-efficiency	Use of R-B is motivated by gains in build efficiency, e.g. Bazel.
	Software-maintenance	The use of R-B is motivated by a need to support software maintenance processes, e.g. Fowler (2010) and Guix.
	System-integrity	SBoM integrity as reason for applying R-Bs. Traceability in an integral component of the motivation.
Tool-certification	A tool or the behaviour of a tool used in a safety-critical environment may need to be certified to support work in that environment.	

Appendix B. Interview protocol

Interviews were conducted using the following eight questions. Question 1 was always asked at the beginning of the interview, and question 8 was always the final question. Question 2 is a prompt for the interviewer to follow up Question 1, if the second part was not answered or as reminder to encourage the interviewee to expand on their answer. Questions 3–7 were used to remind the interviewer of topics to try to cover during the interview, and questions when asked were introduced in the context of the conversation as far as possible, and not as an abrupt change of direction. The research objective(s) that responses to each question are expected to contribute to are indicated.

1. Can you describe your work and how you use reproducible builds? (O2 & O3)
2. How do you use reproducible or deterministic build processes? (O2 & O3)
3. What challenges do you face:
 - (a) with the build process? (O2)
 - (b) with other parties upstream and downstream in the supply chain? (O2)
4. Why do you use reproducible builds? (O2 & O3)
5. What other use cases do you see? (O3)
6. Do you see demand from your “customers”? (O2)
7. Do you think there is a business case or business demand for the use of reproducible builds? (O2 & O3)
8. Is there anything that you think I should have asked you about? (O2 & O3)

Acknowledgements The authors are grateful for the stimulating collaboration and support from colleagues and partner organisations. We would also like to thank all those who kindly took part in the interviews reported in this article.

Funding Open access funding provided by University of Skövde. This research has been financially supported by the Swedish Knowledge Foundation (KK-stiftelsen) and participating partner organisations in the LIM-IT project.

Availability of data and material Not applicable

Code availability Not applicable

Declarations

Conflict of interest The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- ACT. (2020). Automated compliance tooling. <https://automatecompliance.org/>, Retrieved: 01 Jul 2022.
- Ågerfalk, P., Deverell, A., Fitzgerald, B., et al. (2005). Assessing the role of OSS in the European secondary software sector: A voice from industry. In: Proceedings of the 1st International Conference on Open Source Software, pp 82–87.
- Alpine Linux. (2020). Alpine Linux. URL <https://alpinelinux.org/>, Retrieved: 01 Jul 2022.
- Apache Maven. (2022). Configuring for reproducible builds. <https://maven.apache.org/guides/mini/guide-reproducible-builds.html>, Retrieved: 01 Jul 2022.
- Bazel. (2020). Bazel — a fast, scalable, multi-language and extensible build system. URL <https://bazel.build/>, Retrieved: 01 Jul 2022.
- Bitcoin Project. (2022). Bitcoin core. URL <https://bitcoin.org/en/bitcoin-core/>, Retrieved: 01 Jul 2022.
- Braa, K., & Vidgen, R. T. (1999). Interpretation, intervention and reduction in the organizational laboratory: A framework for in-context information systems research. *Accounting, Management and Information Technologies*, 9(1), 25–47. [https://doi.org/10.1016/S0959-8022\(98\)00018-6](https://doi.org/10.1016/S0959-8022(98)00018-6)
- Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2), 77–101. <https://doi.org/10.1191/1478088706qp063oa>
- Bressers, J. (2016). Trusting, trusting trust. <http://sobersecurity.blogspot.com/2016/05/trusting-trusting-trust.html>, Retrieved: 01 Jul 2022.
- Chen, R. (2018). Why are the module timestamps in Windows 10 so nonsensical? <https://devblogs.microsoft.com/oldnewthing/20180103-00/?p=97705>, Retrieved: 30 Jun 2020.
- Courtès, L. (2013). Functional package management with Guix. In: Proceedings of ELS 2013 - 6th European Lisp Symposium, pp 4–14. <https://european-lisp-symposium.org/static/proceedings/2013.pdf#page=10>
- Courtès, L. (2017). Code staging in GNU Guix. In: Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. Association for Computing Machinery, New York, NY, USA, GPCE 2017, pp 41–48. <https://doi.org/10.1145/3136040.3136045>
- Courtès, L., & Wurmus, R. (2015). Reproducible and user-controlled software environments in HPC with Guix. In: Euro-Par 2015: Parallel Processing Workshops. Springer International Publishing, Cham, pp 579–591. https://doi.org/10.1007/978-3-319-27308-2_47
- Courtès, L. (2019). Connecting reproducible deployment to a long-term source code archive. <https://guix.gnu.org/blog/2019/connecting-reproducible-deployment-to-a-long-term-source-code-archive/>, Retrieved: 01 Jul 2022.
- Courtès, L. (2020). Guix: Unifying provisioning, deployment, and package management in the age of containers. <https://fosdem.org/2020/schedule/event/guix/>, [Video] Retrieved: 01 Jul 2022.
- de Carné de Carnavalet, X., & Mannan, M. (2014). Challenges and implications of verifiable builds for security-critical open-source software. In: Proceedings of the 30th Annual Computer Security Applications Conference. ACM, New York, NY, USA, ACSAC '14, pp 16–25. <https://doi.org/10.1145/2664243.2664288>
- Dolstra, E., Löh, A., & Pierron, N. (2010). NixOS: A purely functional Linux distribution. *Journal of Functional Programming*, 20(5–6), 577–615. <https://doi.org/10.1017/S0956796810000195>
- Dong, C. (2019). Bitcoin build system security. <https://www.youtube.com/watch?v=I2iShmUTEI8>, [Video] Retrieved: 01 Jul 2022.
- Edge, J. (2019). A backdoor in a popular Ruby gem. URL <https://lwn.net/Articles/785386/>, Retrieved: 01 Jul 2022.
- Egts, D., & Hellekson, G. (2021). Dave & Gunnar show episode 212: Security requires thinking (his monkey, his circus). <https://dgshow.org/212>, Retrieved: 01 Jul 2022.
- Enck, W., & Williams, L. (2022). Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(2), 96–100. <https://doi.org/10.1109/MSEC.2022.3142338>
- European Commission. (2021). Proposal for a regulation of the european parliament and of the council laying down harmonised rules on artificial intelligence (artificial intelligence act) and amending certain union legislative acts. <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52021PC0206>, Retrieved: 01 Jun 2022.
- European Council. (2016). Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/ec (General Data Protection Regulation). <https://eur-lex.europa.eu/eli/reg/2016/679/oj>, Retrieved: 30 Jun 2021.
- Fowler, M. (2010). ReproducibleBuild. <https://martinfowler.com/bliki/ReproducibleBuild.html>, Retrieved: 01 Jul 2022.

- Gallagher, R., & Greenwald, G. (2014). How the NSA plans to infect ‘millions’ of computers with malware. <https://theintercept.com/2014/03/12/nsa-plans-infect-millions-computers-malware/>, Retrieved: 25 Oct 2021.
- GCC. (2020). The C preprocessor: Section 13 environment variables. URL <https://gcc.gnu.org/onlinedocs/cpp/Environment-Variables.html>, Retrieved: 01 Jul 2022.
- Geyer-Blaumeiser, L. (2019). Ensuring open source compliance using Eclipse Foundation technology. https://github.com/Open-Source-Compliance/Sharing-creates-value/blob/master/Presentations/2019_10_22_EclipseConEurope_EnsuringOpenSourceCompliance.pdf, Retrieved: 01 Jul 2022.
- GNU Guix. (2019). GNU Guix — GNU’s advanced distro and transactional package manager. <https://guix.gnu.org/>, Retrieved: 01 Jul 2022.
- Google Cloud. (2020). Binary authorization for Borg: How Google verifies code provenance and implements code identity. <https://cloud.google.com/security/binary-authorization-for-borg>, Retrieved: 01 Jul 2022.
- GREAT AMR. (2019). Operation shadowHammer: A high profile supply chain attack. <https://securelist.com/operation-shadowhammer-a-high-profile-supply-chain-attack/90380/>, Retrieved: 01 Jul 2022.
- Greenberg, A. (2017). Software has a serious supply-chain security problem. <https://www.wired.com/story/ccleaner-malware-supply-chain-software-security/>, Retrieved: 01 Jul 2022.
- Greenberg, A. (2018). A mysterious hacker group is on a supply chain hijacking spree. URL <https://www.wired.com/story/barium-supply-chain-hackers/>, Retrieved: 01 Jul 2022.
- Guix-HPC. (2020). Guix-HPC reproducible software deployment for high-performance computing. <https://hpc.guix.info/>, Retrieved: 01 Jul 2022.
- Hemel, A. (2020). Docker containers for legal professionals. URL https://www.linuxfoundation.org/wp-content/uploads/Docker-Containers-for-Legal-Professionals-Whitepaper_042420.pdf, Retrieved: 01 Jul 2022.
- Hurst, R. (2021). Verifiable design in modern systems. URL <https://security.googleblog.com/2021/07/verifiable-design-in-modern-systems.html>, Retrieved: 29 Jun 2022.
- Ivanović, M., Petrović, G., Just, R., et al. (2019). Code coverage at google. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, pp 955–963. <https://doi.org/10.1145/3338906.3340459>
- Jacomet, L. (2020). Protecting yourself against attacks through the build. URL <https://jfokus.se/jfokus20/talks/302>, [Video] Retrieved: 01 Jul 2022.
- Johnson, E. (2020). H.R.6216 - national artificial intelligence initiative act of 2020. URL <https://www.congress.gov/bill/116th-congress/house-bill/6216>, Retrieved: 01 Jun 2022.
- Kang, Y., Chen, Z., & Wei, R. (2015). XcodeGhost S: A new breed hits the US. https://www.freeeye.com/blog/threat-research/2015/11/xcodeghost_s_a_new.html, Retrieved: 25 Oct 2021.
- Kuhn, B. M., McAffer, J., Sills, M., et al. (2020). Does careful inventory of licensing bill of materials have real impact on FOSS license compliance? URL https://fosdem.org/2020/schedule/event/debate_license_compliance/, [Video] Retrieved: 01 Jul 2022.
- Lamb, C., & Luo, X. (2017). SOURCE_DATE_EPOCH specification. <https://reproducible-builds.org/specs/source-date-epoch/>, Retrieved: 01 Jul 2022.
- Lamb, C., & Zacchiroli, S. (2021). Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software*, 39(2), 62–70. <https://doi.org/10.1109/MS.2021.3073045>
- Levsen, H. (2016). Beyond reproducible builds. making the whole free software ecosystem reproducible and then https://archive.fosdem.org/2016/schedule/event/reproducible_ecosystem/, [Video] Retrieved: 01 Jul 2022.
- Levsen, H., et al. (2019). Overview of various statistics about reproducible builds. URL <https://tests.reproducible-builds.org/debian/reproducible.html>, Retrieved: 01 Jul 2022.
- Linderud, M. (2019). Reproducible builds: Break a log, good things come in trees. Master’s thesis, University of Bergen, <http://bora.uib.no/handle/1956/20411>
- Lundell, B., & Gamalielsson, J. (2017). Collaborative research involving small companies: Experiences from co-production of knowledge for research and practice through use of an action case approach. In: 2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER IP), pp 24–30. <https://doi.org/10.1109/SER-IP.2017.4>
- McDonald, N., Schoenebeck, S., & Forte, A. (2019). Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. Proceedings of the ACM on Human-Computer Interaction 3(CSCW). <https://doi.org/10.1145/3359174>
- Navarro Leija, O.S., Shiptoski, K., Scott, R.G., et al. (2020). Reproducible containers. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’20). ACM, New York, NY, USA. <https://doi.org/10.1145/3373376.3378519>

- Nesbitt, A., & Pounds, A. (2019). The manifest – Episode 14: Debian and reproducible builds with Chris Lamb. <https://manifest.fm/14>, [Audio] Retrieved: 01 Jul 2021.
- NixOS. (2020). NixOS Linux. <https://nixos.org/>, Retrieved: 01 Jul 2022.
- Ohm, M., Plate, H., Sykosch, A., et al. (2020). Backstabber's knife collection: A review of open source software supply chain attacks. In: Proceedings of The 17th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). Springer International Publishing, Cham, LNCS, pp 23–43. https://doi.org/10.1007/978-3-030-52683-2_2
- Peachey, K. (2022). Post office scandal: What the horizon saga is all about. URL <https://www.bbc.com/news/business-56718036>, Retrieved: 30 Jun 2022.
- Perry, M. (2013). Reproducible builds: Part one: Moving beyond single points of failure for software distribution. URL <https://blog.torproject.org/deterministic-builds-part-one-cyberwar-and-global-compromise>, Retrieved: 01 Jul 2022.
- Perry, M., Schoen, S., & Steiner, H. (2014). Reproducible builds: Moving beyond single points of failure for software distribution. https://media.ccc.de/v/31c3_-_6240_-_en_-_saal_g_-_201412271400_-_reproducible_builds_-_mike_perry_-_seth_schoen_-_hans_steiner, [Video] Retrieved: 01 Jul 2022.
- Piotrowski, M. (2018). ReproducibleBuilds. <https://wiki.freebsd.org/ReproducibleBuilds>, Retrieved: 01 Jul 2022.
- Porup, J. M. (2016). How to make Linux more trustworthy. <https://arstechnica.com/information-technology/2016/12/how-to-make-linux-more-trustworthy/>, Retrieved: 01 Jul 2022.
- Pothon, F., & Ochem, Q. (2017). AdaCore Technologies for DO-178C/ED-12C. AdaCore. <http://www.adacore.com/gnatpro-safety-critical/avionics/do178c/>
- Potvin, R., & Levenberg, J. (2016). Why Google stores billions of lines of code in a single repository. *Commun ACM*, 59(7), 78–87. <https://doi.org/10.1145/2854146>
- Ramakrishna, S. (2021). New findings from our investigation of sunburst. <https://orangematter.solarwinds.com/2021/01/11/new-findings-from-our-investigation-of-sunburst/>, Retrieved: 01 Jul 2022.
- Ren, Z., Jiang, H., Xuan, J., et al. (2018). Automated localization for unreproducible builds. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018. ACM, New York, NY, USA, pp 71–81. <https://doi.org/10.1145/3180155.3180224>
- Ren, Z., Liu, C., Xiao, X., et al. (2019). Root cause localization for unreproducible builds via causality analysis over system call tracing. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, ASE '19, p 527–538. <https://doi.org/10.1109/ASE.2019.00056>
- Ren, Z., Sun, S., Xuan, J., et al. (2022). Automated patching for unreproducible builds. In: 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pp 200–211. <https://doi.org/10.1145/3510003.3510102>
- Reproducible Builds. (2022). reprotest. <https://salsa.debian.org/reproducible-builds/reprotest>, Retrieved: 01 Jul 2022.
- Reproducible Builds Project. (2019a). Definitions. <https://reproducible-builds.org/docs/definition>, Retrieved: 25 Oct 2021.
- Reproducible Builds Project. (2019b). Reproducible builds – a set of software development practices that create an independently verifiable path from source to binary code. <https://reproducible-builds.org/>, Retrieved: 25 Oct 2021.
- Reproducible Builds Project. (2022). diffoscope: In-depth comparison of files, archives, and directories. <https://diffoscope.org/>, Retrieved: 01 Jul 2022.
- Riehle, D., Harutyunyan, N. (2019). Open-source license compliance in software supply chains. In: Fitzgerald B, Mockus A, Zhou M (eds) Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability: Communications of NII Shonan Meetings. Communications of NII Shonan Meetings, Springer Singapore, Singapore, chap 5, p 83–95. https://doi.org/10.1007/978-981-13-7099-1_5
- Rousseau, G., Di Cosmo, R., & Zacchiroli, S. (2020). Software provenance tracking at the scale of public source code. *Empirical Software Engineering*, 25(4), 2930–2959. <https://doi.org/10.1007/s10664-020-09828-5>
- RTCA. (2011). DO-178C – Software Considerations in Airborne Systems and Equipment Certification. RTCA Incorporated.
- Rudin, C. (2019). Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1, 206–215. <https://doi.org/10.1038/s42256-019-0048-x>
- Secure Systems Lab. (2022). in-toto: A framework to secure the integrity of software supply chains. <https://in-toto.io/>, Retrieved: 01 Jul 2022.

- Shaulov, M. (2016). Bridging mobile security gaps. *Network Security*, 2016(1), 5–8. [https://doi.org/10.1016/S1353-4858\(16\)30006-X](https://doi.org/10.1016/S1353-4858(16)30006-X)
- Shi, Y., Wen, M., Cogo, F. R., et al. (2021). An experience report on producing verifiable builds for large-scale commercial systems. *IEEE Transactions on Software Engineering* pp 1. <https://doi.org/10.1109/TSE.2021.3092692>, (Early access)
- Smith, J. K. (2011). Security incident on Fedora infrastructure on 23 jan 2011. <https://lists.fedoraproject.org/pipermail/announce/2011-January/002911.html>, Retrieved: 01 Jul 2022.
- Software Heritage. (2019). Software Heritage. <https://www.softwareheritage.org/>, Retrieved: 25 Oct 2021.
- SPDX Workgroup. (2021). Software Package Data Exchange. <https://spdx.org/>, Retrieved: 01 Jul 2022.
- Tapas, N., Longo, F., Merlino, G., et al. (2019). Transparent, provenance-assured, and secure software-as-a-service. In: 2019 IEEE 18th International Symposium on Network Computing and Applications (NCA), pp 1–8. <https://doi.org/10.1109/NCA.2019.8935014>
- Thompson, K. (1984). Reflections on trusting trust. *Communications of the ACM*, 27(8), 761–763. <https://doi.org/10.1145/358198.358210>
- Tor Project. (2022). Tor project: Anonymity online. <https://www.torproject.org/>, Retrieved: 01 Jul 2022.
- Torres-Arias, S., Afzali, H., Karthik Kuppasamy, T., et al. (2019). in-toto: Providing farm-to-table guarantees for bits and bytes. In: 28th USENIX Security Symposium, USENIX Security 2019. USENIX Association, pp 1393–1410. <https://www.usenix.org/system/files/sec19-torres-arias.pdf>
- van der Burg, S., Dolstra, E., McIntosh, S., et al. (2014). Tracing software build processes to uncover license compliance inconsistencies. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. Association for Computing Machinery, New York, NY, USA, ASE '14, pp 731–742. <https://doi.org/10.1145/2642937.2643013>
- Vinet, J., & Griffin, A. (2022) Arch Linux. <https://www.archlinux.org/>, Retrieved: 01 Jul 2022.
- Wang, J., Kuo, T., Li, L., et al. (2020). Assessing and restoring reproducibility of Jupyter notebooks. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Association for Computing Machinery, New York, NY, USA, ASE '20, pp 138–149. <https://doi.org/10.1145/3324884.3416585>
- Wheeler, D. A. (2005). Countering trusting trust through diverse double-compiling. In: 21st Annual Computer Security Applications Conference (ACSAC'05), pp 13–48. <https://doi.org/10.1109/CSAC.2005.17>
- Wheeler, D. A. (2009). Fully countering trusting trust through diverse double-compiling. PhD thesis, George Mason University
- Xiao, C. (2015). More details on the XcodeGhost malware and affected iOS apps. <https://unit42.paloaltonetworks.com/more-details-on-the-xcodeghost-malware-and-affected-ios-apps/>, Retrieved: 01 Jul 2022.
- Yocto Project. (2021) Reproducible builds. https://wiki.yoctoproject.org/wiki/Reproducible_Builds, Retrieved: 01 Jul 2022.
- Zerouali, A., Mens, T., Robles, G., et al (2019) On the relation between outdated Docker containers, severity vulnerabilities, and bugs. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 491–501.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Simon Butler received a PhD in computing from The Open University in 2016. He is an associate senior lecturer at the University of Skövde and is a member of the Software Systems Research Group. His research interests include software engineering, open source software, program comprehension, software development tools and practices, and software maintenance.



Jonas Gamalielsson received a PhD from Heriot Watt University in 2009. He is a senior lecturer at the University of Skövde and is a member of the Software Systems Research Group. He has conducted research related to free and open source software in a number of projects, and his research is reported in publications in a variety of international journals and conferences.



Björn Lundell received a PhD from the University of Exeter in 2001, and leads the Software Systems Research Group at the University of Skövde. Professor Lundell's research contributes to theory and practice in the software systems domain, in the area of open source and open standards related to the development, use, and procurement of software systems. His research addresses socio-technical challenges concerning software systems, and focuses on lock-in, interoperability, and longevity of systems. Professor Lundell is active in international and national research projects, and has contributed to guidelines and policies at national and EU levels.



Christoffer Brax received the MSc degree from the University of Skövde in 2000, and a PhD from Örebro University in 2011. He is a consultant with Combitech AB working in systems engineering, requirements management, systems design and architecture, and IT security. Christoffer has 18 years experience as a systems engineer.



Anders Mattsson received the MSc degree from Chalmers University of Technology, Sweden, in 1989 and a PhD in software engineering from the University of Limerick, Ireland in 2012. He has almost 30 years experience in software engineering and is currently R&D manager for Information Products and owner of the software development process at Husqvarna AB. Anders is particularly interested in strengthening software engineering practices in organizations. Special interests includes software architecture and model-driven development in the context of embedded real-time systems.



Tomas Gustavsson received the MSc degree in Electrical and Computer Engineering from KTH Royal Institute of Technology in Stockholm in 1994. He is co-founder and current CTO of PrimeKey Solutions AB. Tomas has been researching and implementing public key infrastructure (PKI) systems for more than 24 years, and is founder and developer of the open source enterprise PKI project EJBCA, contributor to numerous open source projects, and a member of the board of Open Source Sweden. His goal is to enhance Internet and corporate security by introducing cost effective, efficient PKI.



Jonas Feist received the MSc degree in Computer Science from the Institute of Technology at Linköping University in 1988. He is senior executive and co-founder of RedBridge AB, a computer consultancy business in Stockholm.




Bengt Kvarnström received the MSc degree in Applied Physics and Electrical Engineering from LiU Institute of Technology in Linköping in 1981. Prior to retiring in late 2021, he was a senior systems engineer at Saab Aeronautics and leader of the group responsible for the Saab Processes, Methodology and Tools for software development.



Erik Lönroth holds an MSc in Computer Science and is an executive at Dwellir AB. Formerly the Technical Responsible for the high performance computing area at Scania CV AB, he led the technical development of four generations of super computing initiatives at Scania and their supporting subsystems. Erik frequently lectures on development of super computer environments for industry, open source software governance and HPC related topics.

Authors and Affiliations

Simon Butler¹  · Jonas Gamalielsson¹ · Björn Lundell¹ · Christoffer Brax² · Anders Mattsson³ · Tomas Gustavsson⁴ · Jonas Feist⁵ · Bengt Kvarnström⁶ · Erik Lönroth⁷

Jonas Gamalielsson
jonas.gamalielsson@his.se

Björn Lundell
bjorn.lundell@his.se

Christoffer Brax
christoffer.brax@combitech.com

Anders Mattsson
anders.mattsson@husqvarnagroup.com

Tomas Gustavsson
tomas.gustavsson@primekey.com

Jonas Feist
jonas.feist@redbridge.se

Bengt Kvarnström
bengt@kvarnstrom.eu

Erik Lönroth
erik@dwellir.com

- ¹ School of Informatics, University of Skövde, Högskovlevägen, Box 408, SE-541 28 Skövde, Sweden
- ² Combitech AB, Universitetsvägen 14, SE-580 15 Linköping, Sweden
- ³ Husqvarna AB, Drottninggatan 2, SE-561 82 Huskvarna, Sweden
- ⁴ PrimeKey Solutions AB, Plan A8, Sundbybergsvägen 1, SE-171 73 Solna, Sweden
- ⁵ RedBridge AB, Gamla Brogatan, SE-111 20 Stockholm, Sweden
- ⁶ Saab AB, Bröderna Ugglas Gata, SE-581 88 Linköping, Sweden
- ⁷ Scania CV AB, Vagnmakarvägen 1, SE-151 87 Södertälje, Sweden