Attestable Builds: Compiling Verifiable Binaries on Untrusted Systems using Trusted Execution Environments

Daniel Hugenroth*
dh623@cst.cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Mario Lins*
mario.lins@ins.jku.at
Johannes Kepler University Linz
Linz, Austria

René Mayrhofer
rm@ins.jku.at
Johannes Kepler University Linz
Linz, Austria

Alastair R. Beresford
arb33@cst.cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Abstract

In this paper we present attestable builds, a new paradigm to provide strong source-to-binary correspondence in software artifacts. We tackle the challenge of opaque build pipelines that disconnect the trust between source code, which can be understood and audited, and the final binary artifact which is difficult to inspect. Our system uses modern trusted execution environments (TEEs) and sandboxed build containers to provide strong guarantees that a given artifact was correctly built from a specific source code snapshot. As such it complements existing approaches like reproducible builds which typically require time-intensive modifications to existing build configurations and dependencies, and require independent parties to continuously build and verify artifacts. In comparison, an attestable build requires only minimal changes to an existing project, and offers nearly instantaneous verification of the correspondence between a given binary and the source code and build pipeline used to construct it. We evaluate it by building open-source software libraries—focusing on projects which are important to the trust chain and have proven difficult to be built deterministically. The overhead (42 seconds start-up latency and 14% increase in build duration) is small in comparison to the overall build time. Importantly, our prototype can build complex projects such as LLVM Clang without requiring any modifications to their source code and build scripts. Finally, we formally model and verify the attestable build design to demonstrate its security against well-resourced adversaries.

CCS Concepts

• **Security and privacy → Software security engineering**; **Trusted computing**; *Hardware-based security protocols.*

Keywords

Confidential Computing; Attestation; Supply-Chain; Reproducible Builds; Software Security; Verifiable Builds; Attestable Builds

*These authors contributed equally to this work.

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765128

ACM Reference Format:
Daniel Hugenroth, Mario Lins, René Mayrhofer, and Alastair R. Beresford. 2025. Attestable Builds: Compiling Verifiable Binaries on Untrusted Systems using Trusted Execution Environments. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765128

1 Introduction

Executable binaries are digital black boxes. Once compiled, it is hard to reason about their behavior and whether they are trustworthy. In contrast, source code is easier to inspect. However, few have the ability, resources, and patience to compile all their software from scratch. Therefore, we seek to enable recipients to verify that an artifact has been truthfully built from a given source code snapshot. This challenge has been popularized in the now-famous Turing Lecture by Ken Thompson on "Trusting Trust" [61].

The problem of trusting build artifacts also presents itself in commercial settings where source code is typically not shared. In this context, the source code is the only source-of-truth that can be inspected by the employed engineers and auditors: During code review it is code changes and not binary output that is examined and, likewise, audit reports generally reference repository commits and not the hash of the shipped artifact. Hence, *verifiable source-to-binary correspondence* is also relevant in an enterprise setting. Where this correspondence cannot be verified, defects are difficult to identify—allowing them to spread down the supply chain to many targets.

In recent years, adversaries have successfully targeted build processes. During the 2020 SolarWinds hack, attackers compromised the company's build server to inject additional code into updates for network management system software [67]. As there were no changes to the source code repository, only forensic inspection of the build machines eventually unveiled the malicious change. In the meantime, the software was distributed to many customers in industry and government that relied on it to secure access to their internal networks. The US Cybersecurity & Infrastructure Security Agency (CISA) issued an emergency directive requesting immediate disconnect of all potentially affected products [10].

In 2024 a complex supply chain attack (CVE-2024-3094) against the XZ Utils package was uncovered that allowed adversaries to

compromise vulnerable servers running OpenSSH [36]. A key enabler for this attack is that the maintainers of (open-source) projects utilizing Autoconf often manually create certain build assets (e.g., a configure script), add it to a tarball, and then provide it to the packager, who builds the final artifact. In case of XZ, this tarball contained a malicious asset covertly included by the adversary that was not part of the repository. Here both the maintainer and the packager have opportunity to meddle with the final binary artifact.

Reproducible Builds (R-Bs, §2.2) are the typically proposed solution to address potential discrepancies between source code and compiled binaries. Correctly implemented, R-Bs ensure source-to-binary correspondence by making the build process perfectly deterministic. Thus, they guarantee that the same source code always results in a bit-to-bit identical binary artifact output. This enables independent parties to reproduce binary artifacts, thus verifying that a given source input generated a given output. There are many successful projects that implement R-Bs [46, 48, 51].

However, R-Bs come with their own challenges, requiring time-intensive and therefore costly changes to the build process. These are incurred not just as a one-time cost but as a continuous maintenance burden. Further, for closed-source software, the downstream consumer cannot check if their supplier has correctly applied R-B principles, since they are typically not given access to the required source code. Additionally, even for source-available software, the build process and compiler are often not available, for example due to intellectual property or licensing concerns. In reality, R-Bs only provide effective security benefits when there are independent builders who are continuously verifying that distributed artifacts are identical to their locally built ones.

We propose *Attestable Builds* (A-Bs) as a practical and scalable alternative where R-Bs are infeasible or costly to implement—including as a complement to extend R-B guarantees to consumers who cannot verify R-Bs themselves even if the primary build chain has R-B properties. For this we leverage Trusted Execution Environments (TEEs) to ensure that the build process is performed correctly and is verifiable. Unlike previous generations of TEEs (e.g., Intel SGX, Arm TrustZone), modern TEE implementations (e.g., AMD SEV-SNP, Intel TDX, AWS Nitro Enclaves) support full virtual machines with strong protection against interference by the hypervisor and physical attacks. Whereas this technology is typically used to achieve data confidentiality, in this work we leverage its integrity properties. This has an additional security benefit: Since integrity attacks are inherently active attacks, this limits the window of opportunity for attacks.

A-Bs are compatible with the reality of modern software engineering practices and allow the build process to be performed by an untrusted build service run by an untrusted cloud service provider (CSP), as long as the TEE hardware is trusted. The idea of A-Bs also extends to other computed artifacts beyond compiled binaries: For instance, A-Bs can additionally attest to test results and static analyses for additional guarantees about the artifacts (§7). Table 1 highlights the similarities and differences between R-Bs and A-Bs. We believe that A-Bs would have prevented or substantially mitigated the feasibility of the mentioned SolarWind and XZ Utils attacks and/or helped to detect them more easily (§7.1).

The overall A-B design is simple: First, an open-source machine image boots inside a modern TEE. The TEE guest then downloads the source code repository and commits to a hash of the downloaded files, including build instructions, in a secure manner before executing the build process inside a sandbox. Afterwards, the TEE hardware trust anchor attests to the booted image, the committed hash value, and the build artifact. The resulting attestation certificate is shared alongside the artifact and is recorded in a transparency log. Finally, the recipient of the artifact can inspect the certificate locally and fetch the corresponding entry from the transparency log to verify that a given artifact has been built from a particular source code snapshot.

Table 1: Comparison of Reproducible Builds (R-Bs) and Attestable Builds (A-Bs).

Reproducible Builds	Attestable Builds
⊕ Strong source-to-binary correspondence	
⊖ High engineering effort for both initial setup and ongoing build maintenance	⊕ Only small changes to the build environment needed
	⊕ Cloud service compatible
⊖ Dependencies and tool chain need to be deterministic	● Dependencies and tool chain can be R-B or A-B
⊖ Environment might leak into build process undetected	⊕ Enforces hermetic builds
⊕ Machine independent	⊖ Requires modern CPU
● Requires trusting at least one party and their machine	● Requires trusting the hardware vendor
⊖ Requires open source	⊕ Supports closed source and signed intermediate artifacts
⊕ Can be composed to an anytrust setup (§3.4)	

One important insight of our work is that nested sandboxing is required because of the current shortcoming of hardware based enclaves such as AMD SEV-SNP (or Amazon Nitro). For these, the remote attestation guarantees stop at boot time and they do not provide nested enclaves. That is, while the hardware security primitives guarantee to protect the host environment from interference by the guest VM—and vice versa—in terms of memory confidentiality and integrity, there is no guarantee on the run-time state within each VM enclave. However, A-Bs necessarily process and execute untrusted and potentially malicious code within a running VM. Therefore, nested sandboxing is required to allow treating the build process as an untrusted black box within the VM, which could otherwise compromise integrity assumptions of the output artifacts. These properties cannot currently be achieved by either hardware VM enclaves or containerization alone—only in combination. As such, our work motivates the need for nested enclave support which provide much stronger integrity guarantees than an inner-nested sandbox.

We also present a composition of A-Bs and R-Bs that achieves novel trust properties (§3.4). By executing R-Bs inside enclaves from different TEE vendors, consumers can trust the artifact as long as they trust any of the vendors—without having to explicitly

decide which one they trust, i.e., an any-trust model. In a typical R-B setup, users cannot easily verify the particular build environment of the involved parties and whether they might share common vulnerabilities, e.g. the same backdoored CPU firmware. As such, this work also contributes to a better understanding of R-B setups and presents an approach to strictly improve their guarantees.

Finally, we provide a specific threat model of cloud-based build services in relation to A-Bs and R-Bs (§3.2) highlighting the underlying trust assumptions, adversary models, and threats. Based on this, we then formally model and verify our protocol using Tamarin [1], a security protocol verification tool, to show that A-Bs provide relevant security guarantees (§5).

In this paper, we make the following contributions:

- We present a new paradigm called Attestable Builds (A-Bs) that provides strong source-to-binary correspondence with transparency and accountability. We discuss the shortcomings of alternative approaches and devise a design relying on a sandbox and an integrity-protected observer.
- We describe a novel composition of A-Bs and R-Bs which provides trustworthy provenance for compiled artifacts in a strong any-trust model.
- We implement an open-source prototype to demonstrate the practicality of A-Bs by building real-world software including complex projects like Clang and the Linux Kernel as well as packages that are hard to build reproducibly.
- We evaluate the performance of our system and find that it adds a (mitigable) 42 second start-up cost, which is small compared to typical build durations. It also imposes a performance overhead of around 14% in our default configuration and up to 68% when using hardened sandboxes.
- We provide threat modeling for verifiable build paradigms such as A-Bs based on Confidential Computing and discuss how it can be extended to other tasks such as compliance tests and static analysis.
- We formally verify the system using Tamarin and discuss the underlying trust assumptions required.

2 Background

Attestable builds integrates with modern software engineering and CI/CD patterns (§2.1) and provides an alternative to R-Bs (§2.2). For this we leverage Confidential Computing technology (§2.3) and verifiable logs (§2.4). This section introduces the required background and building blocks.

2.1 Modern software engineering & CI/CD

Modern Software Engineering (SWE) involves large teams that requires efficient mediation of their collaboration aspects through software. Many projects rely on source control management (SCM) software like Git [52] and Mercurial [7]. The underlying repositories are often hosted by online services, such as GitHub [25] or Bitbucket [37]. We call these Repository Hosting Providers (RHPs).

With increasing complexity, Continuous Integration (CI), has become an important component in modern software projects. Every published code change triggers a new execution of the project's CI pipeline that builds, tests, and verifies the new code snapshot.

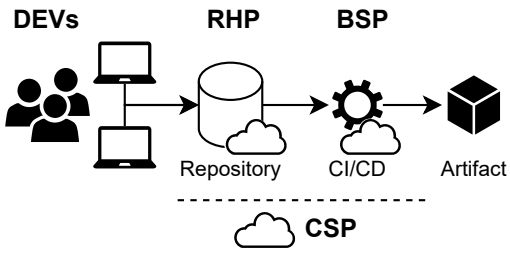

Figure 1: Developers (DEVs) commit to a source code repository at a repository hosting provider (RHP). Changes trigger the CI/CD pipeline at a build service provider (BSP) and generate new binary artifacts. RHP and BSP typically run on servers provided by a cloud service provider (CSP).

In addition, some code changes might trigger a (separate) Continuous Deployment (CD) pipeline which after passing all checks distributes binaries automatically and re-deploys them to the production system. Such CI/CD pipelines are described in configuration files within the source code repository and then executed by online services, build service providers (BSPs), such as Jenkins [49] or GitHub Actions [24]. The latter is an example where the RHP is also a BSP. Our prototype uses GitHub Actions to demonstrate how A-Bs can integrate into existing infrastructure (§4.1).

Both RHPs and BSPs often do not manage their own machines, but use cloud infrastructure provided by cloud service providers (CSPs) such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP). Although there are self-hosted alternatives, such as GitLab [26], even those are often deployed via a CSP. Figure 1 shows the involved parties.

2.2 Reproducible builds (R-Bs)

The use of CI/CD brings many benefits to developers: automated checks ensure that no "broken code" is checked in, builds are easily repeatable since they are fully described in versioned configuration files, and long compile/deploy cycles happen asynchronously. However, they also shift a lot of trust to the RHP, BSP, and CSP. These online services are opaque and any of them can interfere with the build process. Therefore, the conveniently outsourced CI/CD pipeline undermines the trustworthiness of the generated artifacts. This leads to a particularly undesirable situation, as its binary output is hard to inspect and understand. Therefore, trust in the process itself is just as important as trust in its input.

R-Bs have been proposed as a solution to ensure source-to-binary correspondence. The underlying approach is to make the build process fully deterministic such that the same source input always yields perfectly identical binary output. In a project with R-Bs, malicious build servers can be uncovered by repeating the build process on a different machine. Correctly set up, the builds are replicated by independent parties that then compare their results.

However, introducing R-Bs to a software project is challenging [3, 16, 56]. For bit-to-bit identical outputs, the build process needs to be fully described in the committed files and all steps need to be fully deterministic. However, sources of non-determinism are plentiful as outputs can be affected by timestamps, archive metadata,

unspecified filesystem order, build location paths, and uninitialized memory [16, 56].

While many sources of non-determinism can be eliminated with effort and tooling, other steps, such as digital signatures used to sign intermediate artifacts in multi-layered images, cannot easily be made deterministic. This is because typical signature algorithms break when random/nonce parameters become predictable and might leak private key material as a result [29]. For example, consider a build process for a smartphone firmware image that builds a signed boot loader during its process. This inner signature will affect the following build artifacts and is not easily hoisted to a later stage. In other instances, this signing process might happen by an external service or in a hardware security module (HSM) to protect the private key and therefore can never be deterministic.

Critically, for the downstream package to be reproducible, all its dependencies need to be reproducible as well. This also applies for dependencies that are shipped as source code, as R-B is a property of the build system. Facing non-determinism in any of the (transitive) upstream dependencies, a developer either needs to fix the upstream dependency or fork the respective sub-tree. In practice, the verification of having achieved R-B is often done heuristically and newly identified sources of non-determinism can cause a project to loose its status [16]. Despite the challenges, there are large real-world projects that have successfully adopted R-Bs. Examples are Debian [48], NetBSD [27], Chromium [51], and Tor [46]. However, these came at considerable expenses in terms of required upgrades to the build system and on-going maintenance costs [16, 30].

The Debian R-B project stands out due to its scale and highlights the challenges of R-Bs, taking twelve years to produce the first fully reproducible Debian image [18, 38]. A typical challenge is to motivate upstream developers to provide reproducible packages. This led to the introduction of a bounty system offering prioritized inclusion if a build is reproducible [18]. The project's dashboard [13] shows that the number of unreproducible packages dropped from 6.1% (Stretch, released 2017) to 2.0% (Bookworm, released 2023). This suggests that the remaining packages are particularly difficult to convert to R-Bs. Therefore, we picked some of these packages for our practical evaluation (§4.1).

2.3 Confidential Computing

Executing code in a trustworthy manner on untrusted machines is a long standing challenge. Enterprises face this challenge when processing sensitive data in the cloud and financial institutions need to establish trust in installed banking apps. These scenarios require a solution that ensures that the data is not only protected while in-transit or at-rest, but also when in-use. Trusted Execution Environments (TEEs) allow the execution of code inside an *enclave*, a specially privileged mode such that execution and memory are shielded from the operating system and hypervisor. Typically, the allocated memory is encrypted with a non-extractable key such that it resists even a physical attack with probes used to intercept communication between CPU and RAM (and potentially interfere with). Even the hypervisor can only communicate with the enclaves via dedicated channels, e.g., *vsock* or shared memory. However, the hypervisor maintains the ability to pause or stop code execution inside an enclave.

Earlier technologies such as ARM TrustZone [47] and Intel SGX [9] create enclaves on a process level. This requires application developers to rewrite parts of their application using special SDKs so secure functionalities are run inside an enclave. In particular, Intel SGX has proven to be vulnerable to side-channel attacks that allow adversaries to extract secret information from enclaves [5, 41, 58, 62]. It also imposes further practical limitations, such as a maximum enclave memory size and performance overhead.

More recent technologies such as Intel TDX [8] and AMD SEV-SNP [23] boot entire virtual machines (VMs) in a confidential context. This promises to simplify the development of new use-cases as existing applications and libraries can be used with little to no modification. In addition, VMs can be pinned to specified CPU cores, reducing the risk of timing and cache side-channel attacks. AWS Nitro is a similar technology, built on the proprietary AWS Nitro hypervisor and dedicated hardware. The trust model is slightly weaker as the trusted components sit outside the main processor. We choose AWS Nitro for our prototype due to its accessible tooling, but it can be substituted with equivalent technologies.

It is important for the critical software to verify that it is running inside a secure enclave. Likewise, users and other services interacting with critical software need to verify the software is running securely and is protected from outside interference and inspection. This is typically achieved using *remote attestation*. On a high-level, the curious client presents a challenge to the software that claims to run inside an enclave. The software then forwards this challenge to the TEE and its backing hardware who signs the challenge and binds it to the enclave's Platform Configuration Registers (PCRs). The PCRs are digests of hash-chained measurements that cover the boot process and system configuration that claims to have been started inside the TEE [57].

It is typically not possible to run an enclave inside another enclave or to compose these in a hierarchical manner—although new designs are being discussed [4]. This presents a challenge in our case as we need to run untrusted code, i.e. the build scripts stored in the repository, inside the enclave. We work around this technical limitation by sandboxing those processes inside the TEE.

2.4 Verifiable logs

A verifiable log [12] incorporates an append-only data structure which prevents retroactive insertions, modifications, and deletions of its records. In summary, it is based on a binary Merkle tree and provides two cryptographic proofs required for verification. The inclusion proof allows verification of the existence of a particular leaf in the Merkle tree, while the consistency proof secures the append-only property of the tree and can be used to detect whether an attacker has retroactively modified an already logged entry. While such a transparency log is not strictly necessary to verify the attested certificate of an artifact, it adds additional benefits such as ensuring the distribution of revocation notices, e.g., after discovering vulnerabilities or leaked secrets. Artifacts providers can also monitor it to detect when modified versions are shared or their signing key is being used unexpectedly. A central log can also be used to include additional information, such as linking a security audit to a given source code commit (§7).

3 Attestable Builds (A-Bs)

This section introduces the involved stakeholders, the considered threat model, the design of a typical A-B architecture, and how it can be composed with R-Bs.

3.1 Stakeholders

The verifier receives an artifact, e.g., an executable, either directly from a specific BSP or via third-party channels. This could be a user downloading software or a developer receiving a pre-built dependency from a package repository. In general, the verifier does not trust the CI/CD pipeline and therefore wants to verify the authenticity of the respective artifact. **The artifact author**, e.g., a developer or a company, regularly builds artifacts for their project and distributes them to downstream participants. Thus, the system should integrate with existing version control systems hosted by an RHP. The artifact author also does not trust the CI/CD pipeline, as they do not control the involved hardware. Therefore, they need to detect any unauthorized manipulation. **All other stakeholders** (RHP, BSP, CSP, HSP, ...) are untrusted. We assume there are no restrictions on combining multiple roles on one stakeholder, which is the realistic and more difficult set-up as it makes interference less likely to be detected. For example, a self-hosted GitLab operator would take over the role as RHP to manage the source code using git, the BSP by providing build workflows, and the CSP by providing the underlying servers that execute the build steps. Only for the transparency log we require a threshold of honest operators, e.g., in the form of independent witnesses tracking the consistency of the log similar as it is already done in established infrastructure such as Certificate Transparency [32] and Sigstore [44].

3.2 Threat model

The main security objective is to provide an attested build process with strong source-to-binary correspondence guarantees. We do not consider confidentiality or availability as security objectives in A-Bs, assuming that the source code is not inherently confidential and that ensuring availability of relevant components in the build pipeline is the responsibility of the infrastructure provider. However, since the TEEs can also provide confidentiality, A-Bs can be adapted accordingly. Our threat model focuses on the build process as illustrated in Figure 1, describing pipelines where an artifact author publishes code to a repository, which is then built and deployed by the BSP.

3.2.1 Assumptions. We make the following assumptions for our threat model: we assume that the enclave itself is trusted, including the hardware-backed attestation provided by the TEE. We later expand on the intricacies of this statement (§3.2.4) alongside our attack scenarios (§3.6). We assume that the transparency log is trustworthy as potential tampering attempts are detectable. We also assume that the transparency log is protected against split-view attacks by having sufficient witnesses in place.

In this paper, we refer to all components—hardware, firmware, and software—involved directly and indirectly in producing the final artifact as build dependencies. In particular, these dependencies include the TEE firmware, the compilation tool-chain, the image running inside the TEE, and software libraries referenced by the source code and build configuration. Dependencies can consist of or rely on other dependencies recursively. Therefore, in order to make strong provenance claims about the built artifact, all build dependencies must have verifiable provenance, e.g., through R-Bs or A-Bs, to mitigate T3. Otherwise, e.g., a backdoored compilation tool-chain could invalidate the artifact trust assumption (§3.5).

A-Bs rely on benign build dependencies for producing secure artifacts. A malicious build dependency might yield an insecure artifact or otherwise tamper with the build process within the sandbox. However, all build dependencies contribute to either the enclave measurements (firmware, base image) or the source code snapshot (hashes in lockfiles for external dependencies, vendored-in dependencies, ...). Therefore, if a build dependency is later found to be malicious, the affected artifacts can be identified in the transparency log and then revoked.

3.2.2 Adversary modeling. The following list defines relevant adversary models, including information about the respective attack surface, in accordance to the scope of our research.

- A1 **Physical adversary:** Adversary with physical access to hardware, including storage, or the respective infrastructure. We assume that a physical adversary could also be an insider (A3), as our threat model does not distinguish between attacks that require physical access, regardless of whether the attacker is external or internal.
- A2 **On-path adversary (OPA):** An on-path adversary has access to the network infrastructure (e.g., via a machine-in-the-middle [MitM] attack) and is capable of modifying code, the attestation data, or the artifact sent within that network.
- A3 **Insider adversary:** An insider adversary can be a privileged employee working with access to the platform layer such as the hypervisor of the CSP running the VMs or the hosting environment of the BSP. This category of adversary includes malicious service providers. Physical attacks are covered through A1.

3.2.3 Threats. We introduce threats for generic build systems that we considered while designing A-Bs. The following section on architecture explains how A-Bs effectively mitigates these.

- T1 **Compromise the build server:** An adversary (A1, A3) might compromise the build server infrastructure by modifying aspects of the build process, including source code, which could result in a malicious build artifact. This threat addresses all kinds of unauthorized modifications during the build process, such as directly manipulating the source code, the respective build scripts (e.g., shell scripts triggering the build), or parts of the build machine itself, like the OS.
- T2 **Cross-tenant threats:** Any adversary that uses shared infrastructure might use its privilege to temporarily or permanently compromise the host and thus affect subsequent or parallel builds. It also potentially renders any response from the service untrustworthy. This is particularly important for build processes as they generally allow developers to execute arbitrary code.
- T3 **Implant a backdoor in code or assets:** An adversary (A3) might implant a backdoor within the repository through

intentionally incorrect code or within files that are committed as binary assets. For this to be successful the adversary might need to successfully execute social engineering attack to become co-maintainer on an open-source repository. An example of this is the compromise of XZ Utils [36] which we discuss in Section 7. Unlike T1, implanting a backdoor in this manner does not directly compromise the build process itself, but rather is an orthogonal supply chain concern.

T4 **Spoofing the repository:** An adversary might clone an open-source project, introduce malicious modifications, and attempt to make it appear as the original repository as shown in recent attacks [22]. This is similar to typo-squatting of dependencies in package managers [43, 59]. A common mitigation of such threats is the use of digital signatures for signing the artifact. However, an insider adversary (A3) might be able to exfiltrate such a key.

T5 **Compromise build assets during transmission:** An adversary with network access (A2) might compromise build assets (e.g., source code, dependencies, compilation tool-chain, configuration, …) transferred between the parties involved in the build process by intercepting the network traffic.We consider well-resourced adversaries that might issue valid SSL certificates or compromise the servers of any other party. This threat does also include side-loading potentially malicious libraries from external sources.

T6 **Compromise the hardware layer:** An adversary with physical access (A1) might perform classical physical attacks such as interrupting execution, intercepting access to the RAM, and running arbitrary code on the CPU cores that are not part of a secure enclave. This aligns with the threat model of Confidential Computing technologies although they all vary slightly and they do have known vulnerabilities.

T7 **Undermine verification results:** An adversary (A1, A2, A3) can undermine verification results, e.g., authenticity or integrity checks, by manipulating verification data either directly in the infrastructure or while in transit. Similarly, an adversary (A2) might pursue a split-view attack in which some users receive different results for queries against logs.

3.2.4 Confidential Computing. For A-Bs, we require the underlying TEE technology to provide unforgeable remote attestation covering hardware, firmware, and the image running inside the enclave. The security of the attestation is the most critical TEE guarantee as it later allows identifying and disapproving artifacts built in retrospectively-insecure environments (§3.6). In addition, we require strong integrity properties, i.e. the measured enclave code is executed correctly and isolated from the host system. We do not require confidentiality for A-B. So, many attacks targeting confidentiality, including many side-channel attacks, do not impact the provenance guarantees of artifacts built with A-Bs (§6). However, if confidentiality of source code and build configuration is desired, this can be added as an optional security goal.

3.3 Architecture

We designed A-Bs with cloud-based CI/CD pipelines in mind. In particular, such a system can be provided by a BSP who rents infrastructure from an untrusted CSP (see Figure 1). Our design is compatible with different Confidential Computing technologies. While our practical implementation (§4) uses a particular technology, we describe our architecture and its design challenges in general terms (e.g., TEE, sandbox). Figure 2 provides an architectural overview which is described in more detail in this section.

The core unit of an A-B system is the host instance which runs control software, the instance manager, and can start our TEE. Each build request is forwarded to an instance manager which then starts a fresh enclave from a public image inside the TEE. These images are available as open source and therefore have known PCR values that can later be attested to.

The TEE provides both confidentiality and integrity of data-in-use through hardware-backed encryption of memory which protects it from being read or modified—even from adversaries with physical access, the host, and the hypervisor. The enclave uses remote attestation to prove that it has booted a particular secure image in a secure context. These guarantees mitigate T1 and are essential to the integrity of the final attestation. However, it alone is not sufficient, as the build process might manipulate its internal state, and thus the state we are later attesting to. Therefore, we introduce an integrity-protected observer, the *Enclave Client*, that interacts with a sandbox embedded within the TEE.

Once the enclave has booted, it starts the Enclave Client. As it runs inside the TEE, we can assume that it is integrity-protected. The Enclave Client first establishes a bi-directional communication channel with the Instance Manager outside the TEE via shared memory. Through this channel, the Instance Manager provides short-lived authentication tokens for accessing the repository at the RHP and receives updates about the build process.

The Enclave Client then manages a *sandbox* inside the enclave. The sandbox ensures that the untrusted build process (which might execute arbitrary build steps and code) cannot modify the important state kept by the Enclave Client. In particular, we need to protect the initial measurement of the received source code files and build instructions. This mitigates T2. The sandbox optionally captures complete, attested, logs of all incoming and outgoing communication of the build execution, which can help audits and investigations.

Once the sandbox has started, the Enclave Client forwards a short-lived authentication token to the build runner inside the sandbox. The build runner uses the token to fetch both the code and build instructions from the RHP. Since the enclave has no direct internet access, all TCP/IP communication is tunneled via shared memory as well. Upon downloading the source code and instructions, the sandbox computes the commit hash **CT** and reports to the Enclave Client. The commit hash not only covers the content of the code and build instructions, but also the repository metadata. This includes the individual commit messages which can include signatures with the developers private keys [53]. By checking and verifying these during the build steps, the system also attests to the origin of the source code, i.e. the latest developer implicitly signs-off on the current repository state at this commit. This mitigates T5.

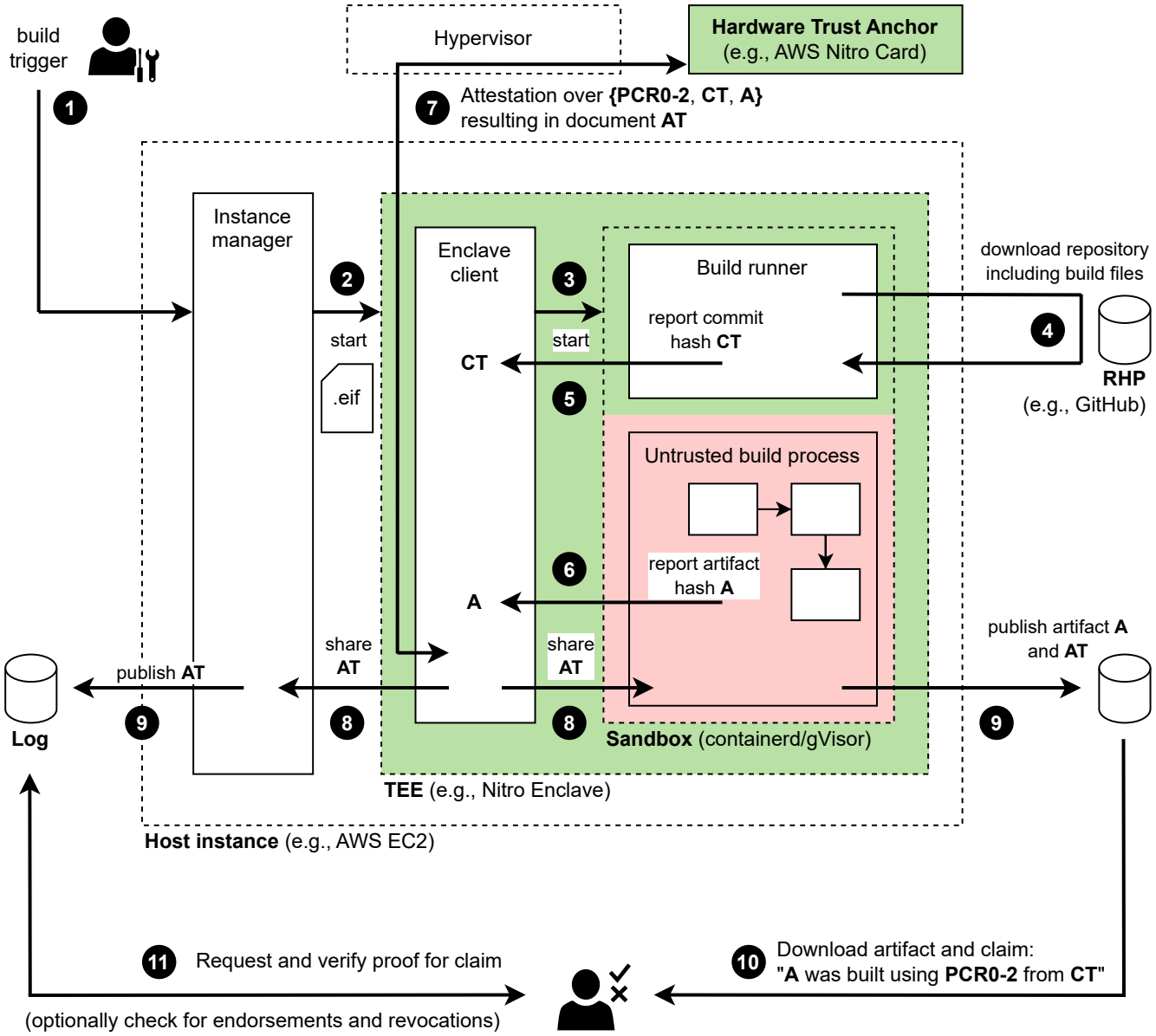

Figure 2: Overview of the protocol steps during build and verification. Dashed borders indicate separate or sandboxed execution environment. Only the TEE and the hardware trust anchor are fully trusted.

❶ The build process is triggered manually or as a result of code changes. Either will cause a webhook call to the Instance Manager. ❷ The Instance Manager starts an fresh enclave from a publicly known `.eif` file with the measurements PCR0-2. ❸ Once booted, the Enclave Client starts the inner sandbox. ❹ The sandbox executes the action runner which fetches the repository snapshot. That snapshot includes both the source code and build instructions. ❺ A hash of the snapshot is reported to the Enclave Client for safeguarding. Now the build process is started which is untrusted. ❻ Once it finishes, the sandbox reports the hash of the produced artifact. ❼ The Enclave Client then requests an attestation document from the Nitro Card covering PCR0-2, the repository snapshot hash, and the artifact hash. ❽ The results are shared with both the build process and the outer Instance Manager. ❾ The build process can now publish the artifact and certificate. And the Instance Manager publishes the attestation. ❿ When a user downloads the artifact, it can contain a certificate specifying how it was build. ⓫ The user can verify this certificate by checking that it is included in the public transparency log.

Once the commit hash has been committed to the Enclave Client, the sandbox starts the build process by executing the build instructions from the repository—and *from that moment we consider the inner state sandbox untrusted.* The sandbox expects the build process to eventually report the path of the artifact that it intends to publish. Once the build process is complete, the sandbox computes the hash **A** of the artifact and forwards it to the Enclave Client. Note: while the inner state of the sandbox is untrusted, the Enclave Client as an integrity-protected observer has safeguarded the input measurements (CT) from manipulation. A ratcheting mechanism ensures that it will only accept CT once at the beginning from the build runner before any untrusted processes are started inside the sandbox. The hash of the artifact (A) can be received from the untrusted build process as it will be later compared by the user against the received artifact.

The Enclave Client then uses the TEE attestation mechanism to request an attestation document **AT** over the booted image **PCR** values (including both the Enclave Client and the sandbox image), the initial input measurement **CT**, and the artifact hash **A**. The response **AT** is then shared with the sandbox, so that the build process can include it with the published artifact, published to the transparency log. Together with proper verification by the client this mitigates T7.

Importantly, the transparency log ensures that revocation notices (e.g., after discovering hardware vulnerabilities) are visible to all users. By requiring up-to-date inclusion proofs for artifacts, the end consumer can efficiently verify that they still considered secure. As such, it lessens the impact of T3 and T6. Furthermore, transparency logs allow the developer to monitor for leaked signing keys. They assure users that observed rotations of signing keys are intentional as they know that developers are being notified about them as well. This mitigates T4.

After completion, the enclave is destroyed. This makes the build process stateless which simplifies debugging and reasoning about its life cycle and helps in mitigating T2, T6. Its stateless nature and the clear control of the ingoing code and build instructions ensures that the build is hermetic, i.e. the build cannot accidentally rely on unintended environmental information. Note that the main build process generally does not require any modifications if it already works with a compatible build runner—it is simply being executed in a sandbox inside an integrity-protected environment. The developer will only need to add a final step to communicate the artifact path and receive the attestation document **AD**.

3.4 Composing A-Bs and R-Bs

We believe that combining our A-Bs and classic R-Bs improves build ergonomics and increases trust. R-Bs can easily consume A-B artifacts and commit to a hash of the artifact similar to lockfiles that are already used by dependency managers such as Rust's cargo and JavaScript's NPM. Similarly, A-Bs can consume R-B artifact and even be independent R-B builders themselves. Due to the attested and controlled environment, existing R-B projects might be able to rely on fewer independent builders when A-Bs are used.

This allows for a setup where the independent builders of an R-B project are distributed across attestable builders running on machines using hardware from different Confidential Computing vendors (see Figure 3). In this setting, the guarantees of the R-B imply an anytrust model that is easily verified. The verifier can use the log to ensure they get a correct build as long as they trust at least one of the Confidential Computing vendors—without having to decide which one. The reader might find it interesting to compare this with how anonymity networks like mix nets and Tor work where traffic is routed through multiple hops and the unlinkability property holds as long as one of them is trusted.

The trust of A-Bs depends on the trust of their build image. While the final artifact (or rather its measurement) is attested to and included in the certificate, we rely on the initial image of the machine embedded in the TEE to ensure the correct and secure execution of the build instructions of the source code snapshot. We believe that R-Bs are important for bootstrapping an A-B system. Even where the base image can be produced using A-Bs, the very first image should be created using R-Bs and bootstrapped from as little code as possible. Projects like Bootstrappable Build [50] lay the foundation for this approach. In the long run, these R-Bs can be executed by attestable builders as described above.

3.5 Build dependencies

The final artifact relies on a number of components that form the Trusted Computing Base (TCB) or are simply direct dependencies specified through the build configuration, e.g., the compilation tool-chain and software libraries. The TCB also comprises the design of the secure hardware, its firmware, the base image, our implementation of the enclave client, and the sandbox.

Similar to software library dependencies, compiler tool-chains are critical to the trustworthiness of the resulting artifact. While there is no widely-available support for enforcing particular tool-chains, we side-step this issue by making the TEE image of our prototype implementation a single large build dependency that includes the compilation tool-chains. Hence the PCR0 measurement covers the compiler tool-chain as well. For a production system, the large image can be made modular with the compilation tool-chain specified in the build configuration. While creating a fully verified TEE image is primarily an engineering concern, we demonstrate that A-Bs can build some of its critical components such as the Linux kernel and the Clang compiler (§4).

The attestation document **AT** can include a reference and cryptographic hash to a full Software Bill of Materials (SBOM). Including

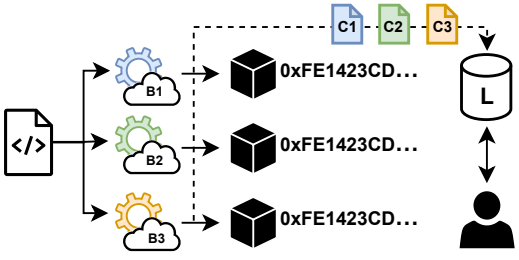

Figure 3: Three attestable builders using different hardware vendors (e.g., Intel, AMD, Arm) perform the same R-B resulting in identical artifacts. The user is then hedged against up to two backdoored TEEs (§3.4).

attestation documents for the individual components in such SBOM document, allows consumers to fully capture the impact of known CVEs against the involved components including the chip firmware and the software running in the enclave. The in-toto standard [14] could be extended for this purpose.

A-Bs do not require the build process to be hermetic, i.e. to be executed without access to the Internet. As long as dependencies are "pinned" using wide-spread support in build tools, e.g. Rust's `Cargo.lock` file, the primary risks of modification of build assets during transmission (T5) are mitigated. Nevertheless, it is considered best-practice for many large software projects to "vendor in" dependencies, i.e. to copy their source code and/or assets into the main repository. This ensures availability and allows for hermetic builds without access to the Internet during the build step. If, optionally, confidentiality of the source code and build configuration is required, the developer might prefer hermetic builds to minimize the risk of leakage. A-Bs are compatible with both pinned dependencies and hermetic builds.

3.6 TEE attacks and their impact on A-Bs

No technology provides absolute security, and CC is no different. There have been recent attacks on popular TEE technologies, including AMD SEV-SNP, that also break integrity properties (§6). However, we find that vulnerabilities affecting TEEs are generally fixed by the chip vendor promptly through firmware updates. Since firmware versions are part of the attestation measurements, the end-user can reject artifacts built in (retrospectively) insecure environments. To our knowledge, there are no successful attacks against AWS Nitro Enclaves, albeit this might be due to the hardware being less accessible to researchers.

A-Bs only require strong remote attestation and integrity, while confidentiality is optional (§3.2.4). Importantly, attacks on confidentiality and integrity differ in one important aspect: integrity attacks are inherently active attacks which in turn implies limitations to the window of opportunity. Therefore, attacks on A-Bs are more difficult for adversaries to achieve since they require persistent access across the fleet and are less opportunistic.

As long as the remote attestation of the most critical measurements, including the CPU firmware, cannot be forged, artifacts that have been built on potentially vulnerable systems can be revoked and rebuilt after the firmware has been patched. In a larger ecosystem, this might trigger rebuilds of large sub-trees of the dependency graph when far-reaching vulnerabilities in core components are discovered. If a TEE ever breaks completely, e.g. the internal signing key for a given model can be extracted, all artifacts from such machines can be revoked and rebuilt with a newer generation of TEEs.

4 Practical evaluation

We implemented the A-B architecture (§3.3) to demonstrate its feasibility and to practically evaluate its performance overhead.

4.1 Implementation

Our prototype uses AWS Nitro Enclaves [63] as the underlying Confidential Computing technology due the availability of accessible tooling. However, it is also possible to achieve similar guarantees with other technologies. For instance, AMD SEV-SNP might offer security benefits due to a smaller Trusted Computing Base (TCB) and we leave this as an engineering challenge for future work.

AWS Nitro Enclaves are started from EC2 host instances and provide hardware-backed isolation from both the host operation system and the hypervisor through the use of dedicated Nitro Cards. These cards assign each enclave dedicated resources such as main memory and CPU cores that are then no longer accessible to the rest of the system. Enclaves boot a `.eif` image that can be generated from Docker images. Creation of these images yields PCR0-2[1] values that can later be attested to.

Since enclaves do not have direct access to other hardware, such as networking devices, all communication has to be done via *vsock* sockets that leverage shared memory. These provide bi-directional channels that we use to (a) exchange application layer messages between the instance manager and enclave client and (b) tunnel TCP/IP access for the build runner to the code repository.

We implemented two sandbox variants using the lightweight container runtime *containerd* and the hardened *gVisor* [21] runtime which has a compatible API. Parameters for the sandbox, such as the short-lived authentication tokens for accessing the repository, are passed as environment variables. Internet access is mediated via Linux network namespaces and results are communicated via a shared log file. We pass only limited capabilities to the sandbox and the runtime immediately drops the execution context to an unprivileged user. gVisor provides additional guarantees by intercepting all system calls. Optionally, this setup can be further hardened using SELinux, seccomp-bpf, and similar.

As we want to demonstrate ease-of-adoption, we integrated with GitHub Actions. The Instance Manager exposes a webhook to learn about newly scheduled build workflows and short-lived credentials are acquired using narrowly-scoped personal access tokens (PAT). Inside the sandbox runs an unmodified GitHub Action Runner (v2.232.0) that is provided by GitHub for self-hosted build platforms. As such, developers only need to export a PAT, add our webhook, and perform minor edits in their `.yml` files (see Appendix C in the extended version of this paper) which include updating the runner name and calling the attestation script.

Most components are written in Rust and we leverage its safety features to minimize the overall attack surfaces and avoid logic errors, e.g., through the use of Typestate Patterns [2]. Our implementation consists of less than 5 000 lines of open source code and is available at: https://github.com/lambdapioneer/attestable-builds.

4.2 Build targets

We demonstrate the feasibility of the A-B approach by building software that appears to be challenging. First, we build five of the still unreproducible Debian packages. We start with a list of all unreproducible packages, choose the ones with the fewest but at least two dependencies (to rule out trivial packages), and then use `apt-rdepends -r` to identify those with the most reverse dependencies, i.e. which likely have a large impact on the build graph. In addition, we add one with more dependencies. This results in the following five packages: *ipxe*, *hello*, *gprolog*, *scheme48*, and *neovim*.

[1] In the AWS Nitro architecture the values PCR0, PCR1, and PCR2 cover the entire `.eif` image and can be computed during its build process.

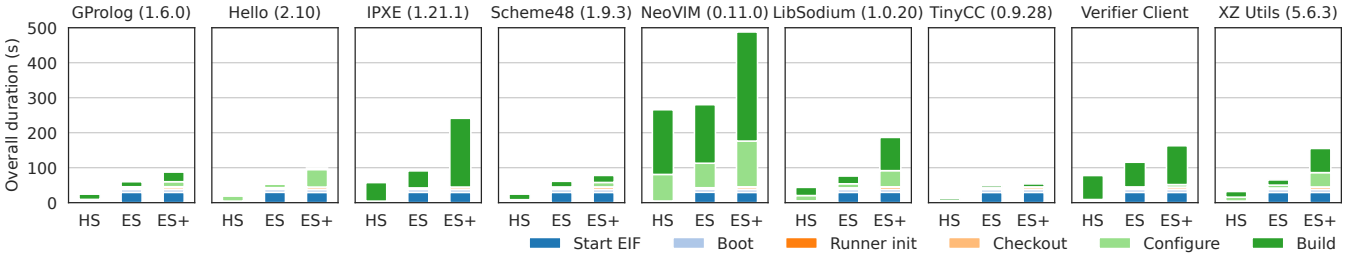

Figure 4: The duration of individual steps for the evaluated projects including the five unreproducible Debian packages and other artifacts. *HS* represents the baseline with a sandbox running directly on the host, *ES* (using containerd) and *ES+* (using gVisor) are variants of our A-B prototype executing a sandboxed runner within an enclave.

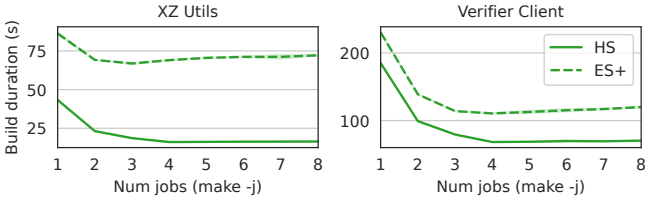

Figure 5: Impact of number of jobs for `make -j` (left) and `cargo build -j` (right) with 4 available CPUs.

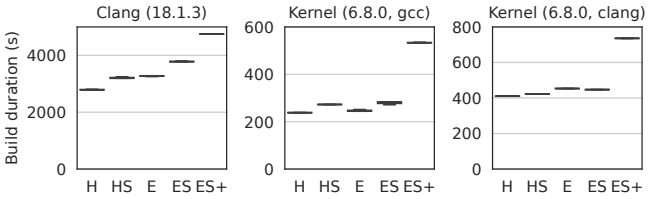

Figure 6: The complex targets *clang* and *kernel* are additionally built without sandboxes on the host *H* and enclave *E*.

Second, we build large software projects including the Linux Kernel (*kernel*, 6.8.0, default config) and the LLVM Clang (*clang*, 18.1.3). These show that our A-Bs can accommodate complex builds and these two artifacts are also essential for later bootstrapping the base image itself, as these are the versions used in Ubuntu 24.04. Finally, we augment this set by including *tinyCC* (a bootstrappable C compiler), *libsodium* (a popular cryptographic library), *xz-utils*, and our own verifier client.

For reproducibility, we include copies of the source code and build instructions in a secondary repository with separate branches for each project. The C-based projects follow a classic configure and make approach and the Rust-based projects download dependencies during the configuration step.

4.3 Measurements

We build most targets on *m5a.2xlarge* EC2 instances (8 vCPUs, 32 GiB). However, for *kernel* and *clang* we use *m5a.8xlarge* EC2 instances (32 vCPUs, 128 GiB). To allow fair comparison between executions inside and outside the enclave, we assign half the CPUs and memory to the enclave. At time of writing, the *m5a.2xlarge* instances cost around $0.34 per hour[2]. We minimize the impact of I/O bottlenecks by increasing the underlying storage limits to 1000 MiB/s and 10 000 operations/s which incurs extra charges.

In order to better understand how the enclave and the sandbox implementations impact performance, we repeat our experiments across three configurations. The *host-sandbox (HS)* configuration runs the GitHub Runner using *containerd* on the host and serves as baseline representing a self-hosted build server. We evaluate two A-B compatible configurations: the *enclave-sandbox (ES)* variant uses the standard *containerd* runtime and the hardened *enclave-sandbox-plus (ES+)* variant uses *gVisor*. For *kernel* and *clang* we additionally include *H* and *E* configurations without sandboxes.

We are interested in the impact of A-Bs on the duration of typical CI tasks. For this we have instrumented our components to add timestamps to a log file. We extract the following steps: *Start EIF* allocates the TEE and loads the `.eif` file into the enclave memory; then the *Boot* process starts this image inside the TEE; subsequently the *Runner init* connects to GitHub and performs the source code *Checkout*; finally, the build file performs first a *Configure* step and then executes the *Build*. We run each combination of build target and configuration three times and report the average.

Figure 4 plots these durations for the unreproducible Debian packages and the additional targets that we have picked (§4.2). For small builds, the overall duration is dominated by the time required to start and boot the enclave. Together these two steps typically take around 37.6 seconds for our 1 473 MiB *.eif* file. These start-up costs can be mitigated by pre-warming enclaves (§7).

For small targets we found that the build duration effectively decreases between *HS* and *ES* configurations. For instance, the NeoVIM build duration (the green bars in Figure 4) drop from 184.9 s (HS) to 167.3 s (ES, -10% over HS). We believe that the enclave is faster because it entirely in memory and therefore mimicks a RAM-disk mounted build with high I/O performance. Again, *gVisor* (ES+) has a large impact and can increase the build times significantly, e.g., NeoVIM takes 311.7 s (ES+, +69% over HS).

The costs for initializing the build runner and checking out the source code are typically less than 9 seconds overall. Even though all IP traffic is tunneled via shared memory using *vsock*, the difference between host-based and enclave-based configurations is small. In fact, for large projects the check-out times sometimes even drops,

[2]For comparison: at the time of writing, the 4-core Linux runner offered by GitHub costs $0.016 per minute ($0.96 per hour).

e.g., *clang* from 148.0 s (HS) to 117.2 s (ES). We believe that the involved Git operations become I/O bound at this size. However, using *gVisor* (ES+) imposes a overhead for the checkout of up-to 2 s for small targets and the checkout of the large *clang* target increases from 117.2 s (ES) to 132.8 s (ES+).

We found that the impact of *gVisor* (ES+) can be lessened by using parallelized builds, e.g., passing the `-j` argument to `make`. Figure 5 shows that ideal number is close to the number of available CPUs. In our case: 4. And while increasing numbers past this point is fine for host-based executions, it has negative impact for *ES+*.

Finally, we build our complex targets *clang* and *kernel* on the larger instance where the TEE is assigned 16 vCPUs and 64 GiB. The larger memory allocation for the TEE increases the *Start EIF* duration from 29.5 s to 46.4 s compared to the smaller instance. Figure 6 shows that there is also a pronounced impact on the build duration. For example, *clang*'s build time increased from 54 minutes (HS) to 63 minutes (ES, +18%) or 79 minutes (ES+, +48%).

For our overall overhead numbers we build all nine small targets and the two large targets back-back. With the baseline configuration *HS* this takes 1h22m. For A-Bs this increases to 1h34m (ES, +14%) and 2h14m (ES+, +62%). These numbers exclude the average start and boot overhead of 42.1s.

5 Formal verification using Tamarin

We use Tamarin [1], a security protocol verification tool, to formally model and verify the underlying protocol of A-Bs. In Tamarin, *facts* represent states of a party involved in a protocol. Thus, we can use facts to describe how the components of our system can interact with each other. Tamarin allows two types of facts: a linear fact that can be consumed only once as it contributes to the system state, and a persistent fact that can be consumed multiple times. A fact in Tamarin is written in the form of $F(t_1..t_n)$, where F is the name of the fact and t_i the value of the current state. We use some built-in facts in Tamarin, like $Fr(x)$, $In(..)$, and $Out(..)$. The $Fr(x)$ fact generates a fresh random value and the $In(..)$ and $Out(..)$ facts are used to receive and send something from and to an adversary-controlled network, respectively.

Tamarin uses multiset rewriting rules (MSR) to describe state transitions. A MSR consists of a name, a left-hand side, an optional middle part, and a right-hand side. The left-hand side defines the facts that needs to be present in order to initiate the MSR. The middle part, called *action fact*, is used to label the specific transition and makes it available for the verification step. The right-hand side describes the state(s) of the outcome.

Finally, we define the security properties to be verified. Tamarin uses *lemmas* to verify both the expected behavior of the protocol and the results of state transitions based on the given *action facts*. Considering the *action facts* including an expected time-dependent relation Tamarin derives traces using first-order logic. The results allow Tamarin to search a trace that contradicts the lemma and thus the security property.

5.1 Security properties

This section outlines various attack categories on security properties used to verify source-to-binary correspondence, including the authenticity of the repository. These attack categories are based on our threat model described in Section 3.2 and we link each category with the respective threat(s) alongside a reference for clarity. The underlying trust assumptions of our threat model (§3.2.1) also apply for the formal model. We model the security properties as formulas in a first-order logic using Tamarin *lemmas*. To verify both protocol behavior and data integrity we utilize *action facts* in the form $F(x_1..x_n)\#i$ where F represents the name of the *action fact*, $x_1..x_n$ the data, and $\#i$ the time variable for the execution. Each subsequent paragraph describes the respective attack category and consists of two proofs: one demonstrating that the specific security property can be successfully compromised when not utilizing A-Bs, and another to ensure that there exists no trace where an adversary would be successful when using A-Bs. We use the function $h(..)$, which represents a hash function and variables c, ct, a, at, ip representing the data: **c**ode, **c**ommi**t**hash, **a**rtifact, **at**testation, and **i**nclusion**p**roof. Appendix D in the extended version of this paper contains the full lemmas of the security properties.

Code manipulation (T1, T2, T6). This attack category examines whether an adversary can successfully manipulate code during the build process. Specifically, this includes compromising code on the build server, attacking shared infrastructure, and considering hardware attacks (assuming the TEE to be trustworthy). Our formal verification begins with proofing that Tamarin can find a trace where an adversary can compromise code c when specific verification controls, used to verify the commit hash ct, are not incorporated. Specifically, this lemma proofs that $\exists\, c, ct : \neg(h(c) = ct)$. For the second proof of this attack category, which includes the verification step, Tamarin does not find any trace where an adversary is able to manipulate code without detection. This proof verifies that $\forall\, c, ct : h(c) = ct$.

Build asset manipulation (T1, T2, T3, T6). The attack category examines whether an adversary can successfully manipulate a build asset (e.g., the artifact) including potentially malicious libraries side-loaded from external sources. Tamarin is able to find a trace where an adversary can successfully compromise a build asset a when specific verification controls, used to verify the inclusion proof ip, are not incorporated. Specifically, this lemma proves that $\exists\, c, ct, at, ip : \neg(h(< ct, h(c), h(c) >) = ip)$. In case of incorporating the verification of the inclusion proof, provided by the transparency log, based on code sent via the adversary network and the attestation at provide by the TEE, Tamarin does not find a trace where an adversary can manipulate a build asset without detection. Specifically, this lemma proves that $\forall\, c, ct, a, at, ip : h(c) = ct \wedge h(< ct, h(a), at >) = ip$.

Build infrastructure manipulation (T1, T2, T6). This attack category focuses on successful attacks in which an adversary is able to compromise the infrastructure environment, i.e. the enclave image. To model this scenario, we transfer the build image through the adversary network so that the adversary can modify it. This analogously covers physical attacks against the machine running the image in an enclave. Thus, our first lemma in this category verifies whether an adversary can provide an attestation document at based on a compromised build image without using the trusted PCR value p to verify the attestation. Specifically, it proves that $\exists\, c, ct, a, p, at : \neg(< c, h(a), p >) = at)$. However, if we include

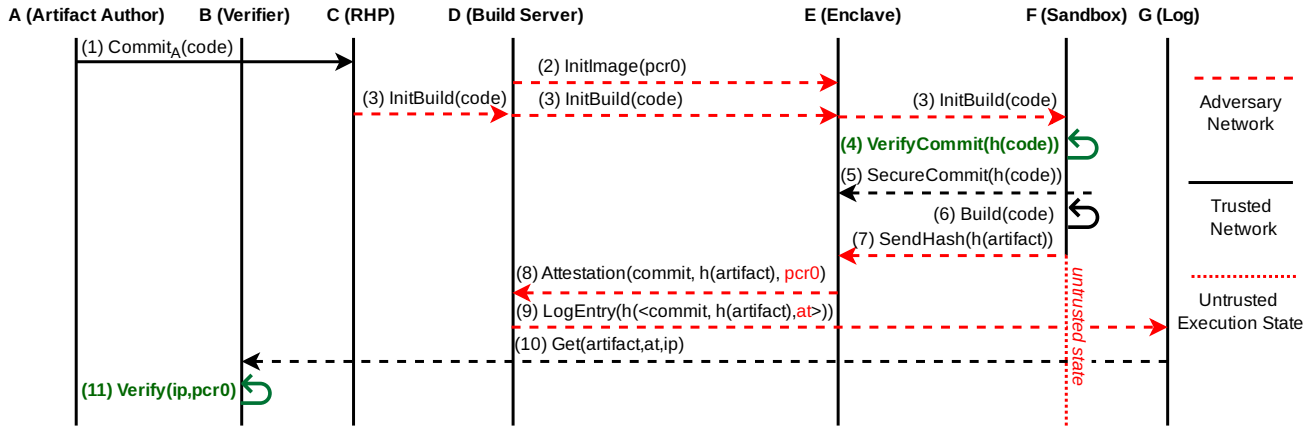

Figure 7: Protocol flow overview of the formal model, illustrating the interactions and data exchanges between system components and adversary channels.

the proper verification in our model, Tamarin does not find any trace where an adversary can use a manipulated build image without detection. The respective proof shows that $\forall\, c, ct, a, p, at : (< c, h(a), p >) = at$.

Repository Spoofing (T4). The last attack category is particularly relevant for spoofing attacks with regards to the repository. An adversary might be able to spoof the repository and to create a valid inclusion proof for a particular commit hash of this repository. In this case, a verifier trying to audit the spoofed repository would get a valid inclusion proof. The first lemma, used to verify whether an adversary can successfully spoof the repository when not verifying the inclusion proof shows that $\exists\, c, ct, a, at, ip : \neg(h(< h(c), h(a), at >) = ip)$. To prevent such spoofing attacks, the artifact author also needs to verify the corresponding inclusion proof according to the trustworthy reference r. Thus, the second lemma proves that $\forall\, c, ct, a, at, ip, r : h(c) = ct \wedge r = ip$.

6 Related work

The challenge of building software artifacts and distributing them in a trustworthy manner has been known for more than 50 years. A report on the Multics system by the US Air Force from 1974, was one of the first to present the idea of a compiler trap door [28]. Ken Thompson popularized the theme of "Trusting Trust" in his Turing Award Lecture in 1984—stating that no amount of source code scrutiny can protect against malicious build processes [61]. In his examples he discusses the implication of a malicious compiler that can introduce a vulnerability in a targeted output binary and preserves this behavior even when it compiles itself from clean source code. David Wheeler suggests Diverse Double-Compiling (DDC) as a practical solution where one uses a trusted compiler to verify the truthful recompilation of the main compiler [64]. However, this leaves open the question on how to arrive at such a trusted compiler as well as to ensure a trustworthy environment to run the proposed steps in. Projects like Bootstrappable Builds discuss approaches to build modern systems from scratch using minimal pre-compiled inputs [50].

The trusted compiler issue can be addressed by having R-Bs and relying either on diverse environments under a at-least-one-trusted assumption or trusting the local setup. The inherent challenges are discussed in academic literature for both individual tools and the overall environment [11, 30]. More papers include industry perspectives on business adoption [3], experience reports for large commercial systems [56], and importance and challenges as perceived by developers [16]. In addition, there has been work aiming at making build environments and tools more deterministic [19, 42, 66].

While deterministic builds aid verification, it also means that the exact same code will be deployed to each target system. This can help attackers since the context of vulnerable code, e.g. register assignments and code pointers, will be exactly the same for each target—potentially also allowing extensive local experiments in the case of generally available software to fine-tune attacks. "Software Diversity" aims at removing this predictability from the generated artifacts by including randomized variation during compilation, linking, and execution stages [31]. A-Bs can support software diversification during the compilation and linking phase since it allows for non-determinism, while R-Bs cannot. However, all approaches are compatible with run-time diversification techniques such as Address Space Layout Randomization (ASLR).

Similar to our approach of using *Confidential Computing* (CC) for providing integrity, Russinovich et al. introduce the idea of Confidential Computing Proofs (CCP) as a more scalable alternative to Zero Knowledge Proofs which rely on heavy and slow cryptography [54]. A-Bs can be seen as a form of CCP that is persisted using a transparency log. Meng et al. propose the use of TPMs in software aggregation to reduce the size of hard-coded lists of trusted binary artifacts [39], but their work lacks a security model and does not generalize to cloud-based CI/CD with untrusted build processes. Others also identified the challenges and opportunities of Confidential Computing as a Service (CCaaS) and our deployment model is inspired by the work by Chen et al. [6].

Trust of pre-built dependencies is key for *supply chain security* and software updates. The framework Supply-chain Levels for

Table 2: The existing SLSA levels L0–L3 adapted from [15] and possible new L4 levels for A-Bs and R-Bs.

	Requirements & focus
L4	Attestable build
	⇀ Attested trust in builder
L4	Reproducible build
	⇀ Verifiable trust in builder
L3	Hardened build platform
	⇀ Tampering during the build
L2	Signed provenance from a hosted build platform
	⇀ Tampering after the build
L1	Provenance showing how the package was built
	⇀ Mistakes, documentation
L0	n/a

Software Artifacts (SLSA) provides helpful threat-modeling and taxonomy to discuss guarantees provided by different systems [15]. Both R-Bs and A-Bs could be adopted as a new level L4 (see Table 2). Frameworks like SLSA become particularly valuable when integrated with codified descriptions such as the in-toto standard [14] CHAINIAC demonstrates how to transparently ship updates using skipchains and verified builds [45].

Sigstore provides an ecosystem [44] to sign and verify artifacts. The authentication certificate together with the artifact hash and the signature is then logged in a transparency log, allowing later verification of a downloaded artifact. Both A-B and the Sigstore project incorporate a transparency log for end-to-end verification. Sigstore makes the signature process verifiable, while we use a transparency log to store metadata about the attested build.

No technology provides absolute security and, in recent years, various researchers have been able to break the security guarantees of TEEs. Since A-Bs do not require confidentiality, many attacks [5, 17, 20, 33–35, 40, 65, 68], including classical side-channel attacks, do not affect its security properties. However, A-Bs rely on strong integrity protection provided by the underlying TEE. To the best of our knowledge, there are no successful attacks that compromise the integrity guarantees of AWS Nitro Enclaves. Appendix B in the extended version of this paper discusses three recent attacks against comparable TEEs.

7 Deployment consideration

Going beyond executable binaries. In this paper we focus on executable binary artifacts that are given to verifiers, e.g., a user downloading new software from the Internet. However, we can also attest other build process outputs. One natural area are supply-chains of software libraries. In such a system, each dependency is built in an attestable manner and the downstream builders verify each included dependency. Since this verification step is part of the attested build process, trust spreads transitively. A-Bs can also attest non-binary artifacts. Examples include results of a vulnerability scanning program (an artifact is secure) or accuracy scores of a benchmark that is run in CI against the built artifact (an artifact meets a certain standard). Another compelling application of the A-B paradigm is its use as part of an issuing authority, e.g., an SSL provider who needs to perform certain checks while creating a new certificate, where trust is an essential aspect.

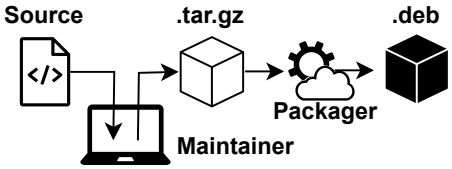

Figure 8: Illustration of the XZ build chain.

Integrating with existing CI/CD systems. Our prototype already integrates with the GitHub Actions CI/CD product using *workflow files* (`.yml`). We found that the required changes are typically less than 10 lines and Appendix C in the extended version of this paper shows a side-by-side comparison of the changes to a typical workflow file. Overall, the developer experience remains the same. A-Bs can be provided by a third-party providing audited base images and run on untrusted CSPs.

Mitigating performance impact. In our evaluation, A-Bs incur a large start-up overhead. However, in practice this can be mitigated by maintaining a number of "pre-warmed" enclaves that are booted, but have not yet fetched any source code. Additionally, as EC2 instances can host a mix of multiple enclaves of various size—given sufficient vCPU and RAM resources—the overall costs can remain low. A load balancer can then redirect build requests to the most suitable ready instance.

Extending the log. In this paper, our transparency log contains entries that link source code snapshots and binary artifacts. However, in a production system these logs can be extended with various types of entries that more holistically capture the security of a given artifact. For example, auditors might provide *SourceAudit* entries signed by their private key to vouch for a given code snapshot and maybe even link them to a set of audit standards published by regulators. Software and hardware vendors might publish *RevocationNotices* when new vulnerabilities are discovered. Based on these, the artifact authors can then ask the independent log monitors to regularly provide compact proofs that attest to the fact that (a) an artifact was built from a given code snapshot, (b) that code snapshot was audited to an accepted standard, and (c) there are no revocation notices affecting this version. The verifier then only needs to check threshold many such up-to-date proofs instead of having to inspect the entire log themselves.

7.1 Case studies

A key aspect of the XZ incident (CVE-2024-3094) [36] was that the adversary added an additional build asset `build-to-host.m4` to the tarball used by the packager to build the final artifact (see Figure 8). Having some pre-generated files (e.g., configure script) is common for projects utilizing *Autoconf* to make the build process easier for others. However, as these build assets are not included in the repository, it is difficult to verify whether they have been generated in a trustworthy manner. Additionally, the concept of R-Bs may not apply as the resulting artifact likely differs when built

on another build host. We believe that A-Bs can add an extra layer of transparency by allowing to verify that the build assets were created in a trustworthy environment based on a specific source code snapshot. Thus, using A-Bs with XZ compels the adversary to use a repository containing all required source code, including the covert `build-to-host.m4` file, to create the tarball that is finally used by the packager.

The SolarWinds hack [67] had a large impact after adversaries successfully compromised a critical supply-chain by implanting a backdoor in a critical software package. The defining aspect of this episode was that the adversaries did not modify the source code in the repository, but were able to compromise the build infrastructure (T1) in a covert manner. Specifically, SUNSPOT was used to inject a SUNBURST backdoor into the final artifact by replacing the corresponding source file during the build process [60]. If A-Bs were used, the change in the PCR values or a failing attestation would have indicated that the build image was modified.

These deployment considerations and potential mitigation for such supply-chain attacks are particularly important for audited, but closed-source firmware. A practical attack demonstration where the authors explain how to engineer a backdoored bitcoin wallet highlights this issue for high-assurance use-cases [55]. We believe that A-Bs can help mitigate such attacks, as the build step itself runs within a trusted and verifiable environment, thus preventing persistent and covert compromise.

8 Conclusion

We presented Attestable Builds (A-Bs) as a new paradigm to provide strong source-to-binary correspondence in software artifacts. Our approach ensures that a third-party can verify that a specific source-code snapshot was used to build a given artifact. A-Bs take into account the modern reality of software development which often relies on a large set of third-parties and cloud-hosted services. We demonstrated this by integrating our prototype with a popular CI/CD framework as part of our evaluation.

Our prototype builds existing projects with no source code changes, and only minimal changes to existing build configurations. We show that it has acceptable overhead for small projects and can also build notoriously complex projects such as LLVM clang. More interesting use-cases are possible, such as attesting non-binary artifacts and building composite systems which also support reproducible builds. Importantly, A-Bs can be pragmatically adopted for difficult gaps in R-B projects as well as an off-the-shelf solution for migrating entire projects.

Acknowledgments

We thank the reviewers and shepherd for their feedback. We also thank Jenny Blessing, Adrien Ghosn, Alberto Sonnino, and Tom Sutcliffe for the valuable discussions and feedback on earlier versions of this paper. All errors remain our own. Daniel is supported by Nokia Bell Labs. This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World and has partially been supported by the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry of Labour and Economy, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

References

[1] Basin, David and Cremers, Cas and Dreier, Jannik and Meier, Simon and Sasse, Ralf and Schmidt, Benedikt. 2025. Tamarin Prover. https://tamarin-prover.com/. Last accessed January 2025.
[2] Cliff L. Biffle. 2024. The Typestate Pattern in Rust. http://cliffle.com/blog/rust-typestate/. Last accessed December 2024.
[3] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Anders Mattsson, Tomas Gustavsson, Jonas Feist, Bengt Kvarnström, and Erik Lönroth. 2023. On business adoption and use of reproducible builds for open and closed source software. *Software Quality Journal* 31, 3 (2023), 687–719.
[4] Charly Castes, Adrien Ghosn, Neelu S Kalani, Yuchen Qian, Marios Kogias, Mathias Payer, and Edouard Bugnion. 2023. Creating Trust by Abolishing Hierarchies. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 231–238.
[5] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPpectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.
[6] Hongbo Chen, Haobin Hiroki Chen, Mingshen Sun, Kang Li, Zhaofeng Chen, and XiaoFeng Wang. 2023. A verified confidential computing as a service framework for privacy preservation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4733–4750.
[7] Mercurial community. 2024. Mercurial Homepage. https://www.mercurial-scm.org. Last accessed November 2024.
[8] Intel Corporation. 2024. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. Last accessed November 2024.
[9] Victor Costan. 2016. Intel SGX explained. *IACR Cryptol, EPrint Arch* (2016).
[10] Cybersecurity & Infrastructure Security Agency. 2021. Emergenc Directive ED 21-01: Mitigate SolarWinds Orion Code Compromise.
[11] Xavier de Carné de Carnavalet and Mohammad Mannan. 2014. Challenges and implications of verifiable builds for security-critical open-source software. In *Proceedings of the 30th Annual Computer Security Applications Conference*. 16–25.
[12] Adam Eijdenberg, Ben Laurie, and Al Cutter. 2015. Verifiable data structures. *Google Research, Tech. Rep* (2015).
[13] Holger Levsen et al. 2025. Overview of various statistics about reproducible builds. https://tests.reproducible-builds.org/debian/reproducible.html. Last accessed April 2025.
[14] The Linux Foundation. 2024. in-toto: A framework to secure the integrity of software supply chains. https://in-toto.io/. Last accessed November 2024.
[15] The Linux Foundation. 2025. Safeguarding artifact integrity across any software supply chain. https://slsa.dev/. Last accessed April 2025.
[16] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. 2023. It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. In *2023 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1527–1544.
[17] Stefan Gast, Hannes Weissteiner, {Robin Leander} Schröder, and Daniel Gruss. 2025. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *Network and Distributed System Security (NDSS) Symposium 2025*. https://www.ndss-symposium.org/ndss2025/ Network and Distributed System Security Symposium 2025 : NDSS 2025, NDSS 2025 ; Conference date: 23-02-2025 Through 28-02-2025.
[18] Paul Gevers. 2023. Bits from the Release Team: Cambridge sprint update. https://lists.debian.org/debian-devel-announce/2023/12/msg00003.html. Last accessed April 2025.
[19] Maria Glukhova et al. 2017. Tools for ensuring reproducible builds for open-source software. (2017).
[20] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.
[21] The gVisor Authors. 2025. gVisor Homepage. https://gvisor.dev/. Last accessed January 2025.
[22] Jossef Harush. 2025. Large Scale Campaign Created Fake GitHub Projects Clones with Fake Commit Added Malware. https://checkmarx.com/blog/large-scale-campaign-created-fake-github-projects-clones-with-fake-commit-added-malware/. Last accessed January 2025.
[23] Advanced Micro Devices Inc. 2024. AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/en/developer/sev.html. Last accessed November 2024.
[24] GitHub Inc. 2024. GitHub Actions: automate your workflow from idea to production. https://github.com/features/actions. Last accessed November 2024.

[25] GitHub Inc. 2024. GitHub Homepage. https://github.com/. Last accessed November 2024.

[26] GitLab Inc. 2024. GitLab Homepage. https://about.gitlab.com/. Last accessed November 2024.

[27] The NetBSD Foundation Inc. 2024. NetBSD fully reproducible builds. https://blog.netbsd.org/tnf/entry/netbsd_fully_reproducible_builds. Last accessed November 2024.

[28] Paul A Karger and Roger R Schell. 2002. Thirty years later: Lessons from the multics security evaluation. In *18th Annual Computer Security Applications Conference, 2002. Proceedings.* IEEE, 119–126.

[29] Jonathan Katz and Yehuda Lindell. 2007. *Introduction to modern cryptography: principles and protocols.* Chapman and hall/CRC.

[30] Chris Lamb and Stefano Zacchiroli. 2021. Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software* 39, 2 (2021), 62–70.

[31] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 276–291.

[32] Ben Laurie. 2014. Certificate transparency. *Commun. ACM* 57, 10 (2014), 40–46.

[33] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17).* 557–574.

[34] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP).* 337–351. doi:10.1109/SP46214.2022.9833768

[35] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21).* 717–732.

[36] Mario Lins, René Mayrhofer, Michael Roland, Daniel Hofer, and Martin Schwaighofer. 2024. On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from XZ. *arXiv preprint arXiv:2404.08987* (2024).

[37] Atlassian Pty Ltd. 2024. Bitbucket – Git solution for teams using JIRA. https://bitbucket.org/product/. Last accessed November 2024.

[38] LWN.net. 2025. Debian bookworm live images now fully reproducible. https://lwn.net/Articles/1015402/, Last accessed April 2025.

[39] Ce Meng, Yeping He, and Qian Zhang. 2009. Remote attestation for custom-built software. In *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, Vol. 2. IEEE, 374–377.

[40] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security.* 1–6.

[41] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE, 1466–1482.

[42] Omar S Navarro Leija, Kelly Shiptoski, Ryan G Scott, Baojun Wang, Nicholas Renner, Ryan R Newton, and Joseph Devietti. 2020. Reproducible containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 167–182.

[43] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2023. Beyond typosquatting: an in-depth look at package confusion. In *32nd USENIX Security Symposium (USENIX Security 23).* 3439–3456.

[44] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. 2022. Sigstore: Software Signing for Everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22).* Association for Computing Machinery, New York, NY, USA, 2353–2367. doi:10.1145/3548606.3560596

[45] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. CHAINIAC: Proactive Software-Update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17).* 1271–1287.

[46] Mike Perry and The Tor Project. 2024. Deterministic Builds Part One: Cyberwar and Global Compromise. https://blog.torproject.org/deterministic-builds-part-one-cyberwar-and-global-compromise/. Last accessed November 2024.

[47] Sandro Pinto and Nuno Santos. 2019. Demystifying ARM TrustZone: A comprehensive survey. *ACM computing surveys (CSUR)* 51, 6 (2019), 1–36.

[48] Debian Project. 2024. Reproducible Builds. https://wiki.debian.org/ReproducibleBuilds/About. Last accessed November 2024.

[49] Jenkins project. 2024. Jenkins Homepage. https://www.jenkins.io/. Last accessed November 2024.

[50] The Bootstrappable Builds project. 2024. Bootstrappable Builds. https://bootstrappable.org/. Last accessed November 2024.

[51] The Chromium Project. 2024. Deterministic builds. https://chromium.googlesource.com/chromium/src/+/HEAD/docs/deterministic_builds.md. Last accessed November 2024.

[52] The Git project. 2024. Git Homepage. https://git-scm.com. Last accessed November 2024.

[53] The Git project. 2024. Git Tools Signing your work. https://git-scm.com/book/ms/v2/Git-Tools-Signing-Your-Work. Last accessed January 2025.

[54] Mark Russinovich, Cédric Fournet, Greg Zaverucha, Josh Benaloh, Brandon Murdoch, and Manuel Costa. 2024. Confidential Computing Proofs: An alternative to cryptographic zero-knowledge. *Queue* 22, 4 (2024), 73–100.

[55] Adam Scott and Sean Andersen. 2024. Engineering a backdoored bitcoin wallet. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24).* USENIX Association, Philadelphia, PA, 89–100. https://www.usenix.org/conference/woot24/presentation/scott

[56] Yong Shi, Mingzhi Wen, Filipe R Cogo, Boyuan Chen, and Zhen Ming Jiang. 2021. An experience report on producing verifiable builds for large-scale commercial systems. *IEEE Transactions on Software Engineering* 48, 9 (2021), 3361–3377.

[57] Gary Simpson, Amy Nelson, Shiva Dasari, Ken Goldman, Nayna Jain, Jiewen Yao, Qin Long, Robert Hart, Ronald Aigner, and Dick Wilkins. 2019. *TCG PC Client Specific Platform Firmware Profile Specification.* Technical Report. Trusted Computing Group. Version 1.04, https://trustedcomputinggroup.org/resource/pc-client-specific-platform-firmware-profile-specification/. Last accessed November 2024.

[58] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. 2019. Microscope: Enabling microarchitectural replay attacks. In *Proceedings of the 46th International Symposium on Computer Architecture.* 318–331.

[59] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending against package typosquatting. In *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14.* Springer, 112–131.

[60] CrowdStrike Intelligence Team. 2021. SUNSPOT: An Implant in the Build Process. (2021). https://www.crowdstrike.com/en-us/blog/sunspot-malware-technical-analysis/. Last accessed January 2025.

[61] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (1984), 761–763.

[62] Stephan Van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX fails in practice. https://sgaxe.com/files/SGAxe.pdf. Last accessed November 2024.

[63] Amazon web services. 2024. AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/. Last accessed December 2024.

[64] David A Wheeler. 2005. Countering trusting trust through diverse double-compiling. In *21st Annual Computer Security Applications Conference (ACSAC'05).* IEEE, 13–pp.

[65] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. 2024. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24).* Association for Computing Machinery, New York, NY, USA, 79–93. doi:10.1145/3658644.3690230

[66] Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R Cogo, and Zhen Ming Jiang. 2022. Towards build verifiability for java-based systems. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice.* 297–306.

[67] Kim Zetter. 2023. The Untold Story of the Boldest Supply-Chain Hack Ever. *Wired* (2023).

[68] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *33rd USENIX Security Symposium (USENIX Security 24).* USENIX Association, Philadelphia, PA, 1135–1151. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-ruiyi

A Extended paper version

The extended paper version is available at: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-1002.html. Its additional appendices contain: a screenshot of the GitHub Action CI running our prototype, a summary of attacks against Confidential Computing technologies, a sample GitHub integration YAML listing, the lemmas used for our formal verification, a sample Tamarin attack trace, additional plots for the practical evaluation, and tabular results. The main text of the extended paper only differs from this paper where it references these additional pieces of information.