Semantic/Deep SBOMs

Lennart Beringer (Princeton University, eberinge@cs.princeton.edu)
David Naumann (Stevens Insititute of Technology, dnaumann@stevens.edu)

Position paper for area 5, Guidelines for software integrity chains and provenance

The EO envisions – among other mechanisms and policies – the creation of Software Bill of Material (SBOM) frameworks that constitute (according to the glossary in Sec. 10(j)) a *...formal record containing the details and supply chain relationships of various components used in building software.* According to the description in Sec. 4(e), these SBOMs are expected to contain not only accurate and up-to-date provenance information of components, but evidence of audits and analyses pertaining to vulnerabilities, ascertain conformity with secure software development practices and disclosure policies, and are provided to a purchaser *directly or by publishing it on a public website.*

These requirements cover *provenance* (chain of custody) and *pedigree* (history of how it was produced) of software[1]. Both aspects are *administrative* and can be captured by metadata, certified using cryptographic means. Indeed Blockchain technologies are finding rapid deployment to trace production histories and logistics chains in BOMs for physical goods. However, earlier in Section 4, the EO states *The development of commercial software often lacks transparency, sufficient focus on the ability of the software to resist attack, and adequate controls to prevent tampering by malicious actors. There is a pressing need to implement more rigorous and predictable mechanisms for ensuring that products function securely, and as intended.* Thus, the overall goal includes expectations that concern functionality – an *inherent* property whose satisfaction is independent of how the code was obtained or produced and can thus not be (purely) covered by metadata.

Extending administrative SBOMs with additional information (including evidence pertaining to code-inherent properties) appears beneficial along at least the following axes:

architectural information: improving on linear catalogs of (software) ingredients, current SBOMs already recognize the nested structure of software by supporting or envisioning tree-shaped or otherwise hierarchically structured relationships between software elements. Taking this one step further, we argue that such structure be extended to capture architectural arrangements as modeled by languages such as AADL [8], and that additional glue code that schedules execution, reformats data streams, or otherwise orchestrates invocation of individual components be included. Indeed, in the age of agile methods, software is routinely obtained not *de nuovo* but composed from existing packages, libraries, or microservices. Knowing *in which manner* a large system relies on a potentially compromised component is as important as knowing whether the component is used at all. Furthermore, the gluecode itself may contains vulnerabilities.

deployment constraints: in addition to routine metadata, a system's SBOM entry should include any information that is necessary to deploy the software and reproduce it: libraries, OS-version, compilers (with invocation flags), and potentially processor information or hardware configurations. These pieces of information are routinely maintained in build scripts, package managers, CI pipeline configurations, or middleware orchestrations. To increase the precision, coarse-grain version numbers should ideally be replaced by (hashes of) specific commits in code repositories. Ultimately, we note that each ingredient itself constitutes an SBOM entry: for example, a compromised compiler affects potentially any component it has been deployed on. Hence, SBOM frameworks must naturally be higher-order, and containment information must be efficiently searchable in both direction (contains/uses versus is-contained-in/is-used-by).

[1] Our use of the words *provenance* and *pedigree* follows Robert A. Martin's slides [12].

validation information: evidence about testing procedures (including test vectors, compiler flags, parameter seeds or anything else that's needed to reliably reproduce results), vulnerability detection tools (with links to the exact set of virus signatures used), assessment tools regarding adherence to coding guidelines, and other static analyses need to be included for each component, so that an integrator can judge the maturity of a component.

Overall, this dimension concerns a component's *bill of health*, which can in principle be considered at arbitrary levels of precision, w.r.t. domain-specific observer/attack models, and may include robustness requirements against side channel attacks (timing, EM radiation, cache behavior, ...). But the tools themselves must again in principle be represented as SBOM entries, reinforcing the need for a recursive / higher-order framework.

functional refinement: when a component is compromised, an SBOM consumer needs to quickly identify possible remedies. To help evaluating potential alternatives, it may be useful to maintain application-specific refinement or substitution orders w.r.t. full functional correctness, conditional equivalence, information flow policies, or other approximate abstractions.

Given these multi-dimensional requirements and the goal that at least some SBOM claims be unforgeable, we expect expressive SBOM frameworks will combine nominal inventories with code repositories and easily searchable component databases, expressive reasoning capabilities, cryptographic machinery, and software analysis and verification tools.

Building on notions such as formalized assurance cases [9, 10, 16], we argue that appropriate conceptual frameworks for understanding SBOMs are higher-order logics or expressive type theories, implemented in interactive proof assistants that embed declarative programming languages. A recent example for a logical system that builds on semi-automatic verification to construct a machine-checkable source-level counterpart of assembly-level linking is VSU [2]. Implemented in the Coq proof assistant and targeting software written in C, VSU extends the Verified Software Toolchain (VST, [5]) to separate compilation units, respecting hierarchical dependencies and supporting parametric module specifications. VST/VSU's notion of specification subsumption [3] permits functional correctness specifications to be abstracted to memory safety or other more lightweight guarantees. This enables e.g. whole-program safety to be obtained compositionally, in a scalable fashion: for most components, safety is established using mostly-automatic static analysis, use of a safe language subset, or via code synthesis from a higher-level language. But some components – including those that provide cryptographic functionality, memory management, or other protection mechanisms (including SBOM validation itself!)– need to be provably correct, not just safe; for these, the full power of VST is available [1, 4]. Integrating support for metadata as discussed above into VSU could yield a foundational basis for *deep* SBOM formats, in which Merkle trees or other blockchain techniques provide cryptographic attestation that oftentimes renders inspecting or replaying proofs unnecessary.

More limited in their capability are Datalog-based frameworks such as the Evidential Tool Bus [7]. These are easier to develop and deploy, and can in principle be combined with proof-carrying-code [15] techniques that certify non-functional code-inherent properties (memory safety, information flow, limited resource consumption,...) of low level code when source code cannot be made available.

To be deployable in application areas envisioned by current NTIA-led SBOM efforts [13, 14] (although not using the presently envisioned formats), deep SBOMs will likely also need to support for incremental builds [6] and dynamic software update [11].

As an example for the necessity to support expressive reasoning, we note that publishing of SBOMs on public websites (as envisioned by the EO) not only enables intended consumers to quickly decide whether they are affected by a new vulnerability. It also benefits attackers, who learn where/how a compromised component is used, can hence design secondary attacks more easily, and receive a readily usable list of potential ransom victims. Thus, visibility of SBOM information may need to come with restrictions / zero-trust / privilege mechanisms; this needs to be analyzed when an SBOM's trust and attack model is defined.

Ultimately, we envision SBOMs to become unified with BOMs for physical artifacts, mirroring the ever tighter integration of software into robotics, IoT, or CPS systems on one hand, and the ever more detailed computational modeling of physical deceives and their application environments, i.e. the construction of digital twins, on the other.

References

[1] A. W. Appel and D. A. Naumann. Verified sequential malloc/free. In C. Ding and M. Maas, editors, *ISMM '20: 2020 ACM SIGPLAN International Symposium on Memory Management, ISMM 2020, virtual [London, UK], June 16, 2020*, pages 48–59. ACM, 2020.

[2] L. Beringer. Verified software units. In N. Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021*, volume 12648 of *LNCS*, pages 118–147. Springer, 2021.

[3] L. Beringer and A. W. Appel. Abstraction and subsumption in modular verification of C programs. In M. H. ter Beek, A. McIver, and J. N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *LNCS*, pages 573–590. Springer, 2019.

[4] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of Openssl HMAC. In J. Jung and T. Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 207–221. USENIX Association, 2015.

[5] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reason.*, 61(1-4):367–422, 2018.

[6] D. Coetzee, A. Bhaskar, and G. C. Necula. A model and framework for reliable build systems. *CoRR*, abs/1203.2704, 2012.

[7] S. Cruanes, G. Hamon, S. Owre, and N. Shankar. Tool integration with the evidential tool bus. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 275–294, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[8] P. H. Feiler, D. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. 2 2006.

[9] S. Foster, Y. Nemouchi, C. O'Halloran, K. Stephenson, and N. Tudor. Formal model-based assurance cases in Isabelle/SACM: An autonomous underwater vehicle case study. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, pages 11–21, New York, NY, USA, 2020. Association for Computing Machinery.

[10] A. Gacek, J. Backes, D. Cofer, K. Slind, and M. Whalen. Resolute: An assurance case language for architecture models. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 19–28, New York, NY, USA, 2014. Association for Computing Machinery.

[11] M. W. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.

[12] R. A. Martin. Software bill of materials. https://csrc.nist.gov/CSRC/media/Projects/cyber-supply-chain-risk-management/documents/SSCA/Spring_2019/8MayAM2.3_Software_Bill_of_Materials_Robert_Martin_05_08_19_clean.pdf, 2019.

[13] National Telecommunications and Information Administration. NTIA Software Component Transparency (web page). https://www.ntia.doc.gov/SoftwareTransparency, 2021.

[14] National Telecommunications and Information Administration. Software Bill of Materials (web page). https://www.ntia.gov/SBOM, 2021.

[15] G. C. Necula. Proof-carrying code. In P. Lee, F. Henglein, and N. D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997.

[16] J. Rushby. Mechanized support for assurance case argumentation. In Y. I. Nakano, K. Satoh, and D. Bekki, editors, *New Frontiers in Artificial Intelligence - JSAI-isAI 2013 Workshops, LENLS, JURISIN, MiMI, AAA, and DDS, Kanagawa, Japan, October 27-28, 2013, Revised Selected Papers*, volume 8417 of *LNCS*, pages 304–318. Springer, 2013.