Verification Across Intellectual Property Boundaries *

Sagar Chaki[1], Christian Schallhart[2], and Helmut Veith[2]

[1] Software Engineering Institute, Carnegie Mellon University, USA
chaki@sei.cmu.edu
[2] Institut für Informatik, Technische Universität München, Germany
schallha@in.tum.de, veith@in.tum.de

Abstract. In many industries, the share of software components provided by third-party suppliers is steadily increasing. As the suppliers seek to secure their intellectual property (IP) rights, the customer usually has no direct access to the suppliers' source code, and is able to enforce the use of verification tools only by legal requirements. In turn, the supplier has no means to convince the customer about successful verification without revealing the source code. This paper presents a new approach to resolve the conflict between the IP interests of the supplier and the quality interests of the customer. We introduce a protocol in which a dedicated server (called the "amanat") is controlled by both parties: the customer controls the verification task performed by the amanat, while the supplier controls the communication channels of the amanat to ensure that the amanat does not leak information about the source code. We argue that the protocol is both practically useful and mathematically sound. As the protocol is based on well-known (and relatively lightweight) cryptographic primitives, it allows a straightforward implementation on top of existing verification tool chains. To substantiate our security claims, we establish the correctness of the protocol by cryptographic reduction proofs.

1 Introduction

In the classical verification scenario, the software author and the verification engineer share a common interest to verify a piece of software; the author provides the source code to be analyzed, whereon the verification engineer communicates the verification verdict. Both parties are mutually trusted, i.e., the verification engineer trusts that he has verified production code, and the author trusts that the verification engineer will not use the source code for unintended purposes.

Industrial production of software-intensive technology however often employs supply chains which render this simple scenario obsolete. Complex products are being increasingly assembled from multiple components whose development is outsourced to supplying companies. Typical examples of outsourced software components comprise embedded controller software in automobiles and consumer electronics [1, 2] as well as Windows device drivers [3]. Although the suppliers may well use verification
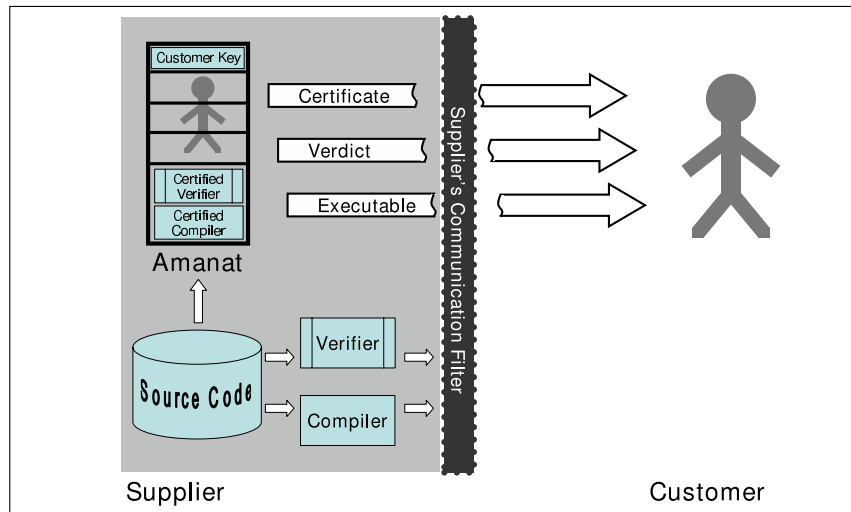

Fig. 1. A High-Level View of the Amanat Protocol

techniques for internal use, they are usually not willing to reveal their source code, as the intellectual property (IP) contained in the source code is a major asset for their company.

This setting constitutes a principal conflict between the *supplier* Sup who owns the source code, and the *customer* Cus who purchases o n l y the executable. While both parties share a basic interest in producing high quality software, it is in the customer's interest to have the source code inspected, and in the supplier's interest to protect the source code. More formally, this amounts to the following basic requirements:

(a) **Conformance.** The customer must be able to validate that the purchased executable was compiled from successfully verified source code.
(b) **Secrecy.** The supplier must be able to validate that no information about the source code other than the verification result is revealed to the customer.

The main technical contribution of this paper is a new cryptographic verification protocol tailored for IP-aware verification. Our protocol is based on standard cryptographic primitives, and provably satisfies both the above requirements with little overhead in the system configuration. Notably, the proposed scheme applies not only to automated verification in a model checking style, but also encompasses a wide range of validation techniques, both automated and semi-manual.

Our solution centers around the notion of an *amanat*. This terminology is derived from the historic judicial notion of amanats, i.e., noble prisoners who were kept hostage as part of a contract. Intuitively, our protocol applies a similar principle: The amanat is a trusted expert of the customer who settles down in the production plant of the supplier and executes whatever verification job the customer has entrusted on him. The supplier accepts this procedure because (i) all of the amanat's communications are subject to the censorship of the supplier, and, (ii) the amanat will never return to the customer again.

It is evident that clauses (i) and (ii) above make it impossible for a human inspector to act as the amanat; instead, our protocol will utilize a dedicated server Ama for this task. The protocol guarantees that Ama is simultaneously controlled by both parties: Cus controls the verification task performed by Ama, while Sup controls the communication channels of Ama. To convince Cus about conformance, the verification tool executed on Ama produces a cryptographic certificate which proves that the purchased executable is derived from the same source file as the verification verdict.

To achieve this goal, we use public key cryptography; the amanat uses the secret private key of the customer, and signs outgoing information with this secret key such that *no additional information can be hidden in the signature*. This enables the supplier to inspect (and possibly block) all outgoing information, and simultaneously enables the customer to validate that the certificate indeed stems from the amanat. Thus, the amanat protocol achieves the two requirements above. Figure 1 presents a high-level illustration of the protocol.

Verification by Model Checking and Beyond. Motivated by discussions with industrial companies, our primary intention for the protocol was to facilitate software model checking across IP boundaries in a B2B setting where the supplier and the customer are businesses. Our guiding examples for this B2B setting have been Windows device drivers and automotive controller software, for which our protocols are practically feasible with state-of-the-art technology.

Software model checking is now able to verify important properties of simply structured code [4–6]. Most notably, SLAM/SDV is a fully automatic tool for a narrow application area, and we expect to see more such tools. Note that SDV has built-in specifications because the device drivers access and implement a clearly defined API. Other tools such as Terminator [7] and Slayer [8] do not require specifications as they are built to verify specific critical properties – termination and memory-safety, respectively. Automotive software is similar to device drivers in that it also accesses standardized APIs.

For less standardized software and more specific properties, it may be necessary for the customer and the supplier to negotiate about the formulation of the specification without revealing the source code. In the course of this negotiation, the supplier can decide to reveal a blueprint of the software, and the amanat can certify the accuracy of the blueprint by a mutually agreed algorithm.

The example of blueprints shows that the amanat protocol is *not restricted* to model checking, because the amanat can run any verification/validation tool whose output does not compromise the secrecy of the source code. For example, in future work and applications, Ama can:

1. apply static analysis tools such as ASTREE [9] and TVLA [10].
2. check the correctness of a manual proof provided by Sup, e.g., in PVS, ISABELLE, Coq or another prover [11].
3. evaluate worst case execution times experimentally [12] or statically [13].
4. generate white box test cases, and execute them.
5. validate that the source code comes with a set of test cases which satisfies previously agreed coverage criteria.
6. check that the source code is syntactically safe, e.g. using LINT.

7. compute numerical quality and quantity measures which are agreed between Sup and Cus, e.g. nesting depth, LOC, etc.
8. compare two versions of the source code, and quantify the difference between them; this is important in situations where Sup claims charges for a reimplementation.
9. check if third party IP is included in the source code, e.g. libraries etc.
10. ensure that certain algorithms are (not) used.
11. check that the source is well documented.
12. ensure a certain senior programmer has put his name on the source code.
13. validate the development steps by analyzing the CVS or SVN tree.
14. ensure compatibility of the source code with language standards.

We note that in all scenarios the code supplier *bears the burden of proof*: either the supplier has to write the source code in such a way that it is accepted by Ama as is, or the supplier has to provide auxiliary information (*e.g.* proofs, command line options, abstraction functions, test cases, etc.) which help the amanat in the verification without affecting correctness.

Security of the Amanat Protocol. In Section 4, we present a cryptographic proof for the *secrecy* and *conformance* of the amanat verification protocol. Stronger than term-based proofs in the Dolev-Yao model, these proofs assure that under standard cryptographic assumptions, randomized polynomial time attacks against the protocol (which may involve e.g. guessing the private keys) can succeed only with negligible probability [14]. The practical security of the protocol is also ensured by the simplicity of our protocol: As the protocol is based on well-known cryptographic encryption and signing schemes, it can be readily implemented.

The IP boundary between the supplier and the customer makes is inevitable that the amanat owns a *secret* unknown to the supplier, namely the private key of the customer; this secret enables the amanat to prove its identity to the customer and to compute the certificate. Consequently, the cryptographic proofs need to assume a system configuration where Ama can neither be reverse-engineered, nor closely monitored by the supplier. Thus, from the point of view of the supplier, Ama is a black box with input and output channels. For secrecy, the supplier requires ownership of Ama to make sure it will not return to the customer after verification. There are two natural scenarios to realize this hardware configuration:

A Ama is physically located at the site of a trusted third party. All communication channels of Ama are hardwired to go through a second server, the communication filter of the supplier, cf. Figure 1.

While scenario A involves a trusted third party, its role is limited to providing physical security for the servers. Thus, the third party does not need any expertise beyond server hosting. For the supplier, scenario A has the disadvantage that the encrypted source code has to be sent to the third party, and thus, to leave the supplier site.

B Ama is physically located at the site of the supplier, but in a sealed location or box whose integrity is assured through (i) regular checks by the customer, (ii) a third party, (iii) a traditional alarm system, or (iv) the use of sealed hardware. All

communication channels of Ama are hardwired to the communication filter of the supplier.

In scenarios B(ii) and B(iii), the third party again plays a very limited role in that it only ensures physical integrity of the amanat. We believe that in our B2B settings, scenario B is realistic. We do not require custom-made hardware, but just a sealed location at the supplier's site, e.g. a locked room. Off-the-shelf hardware ensures that neither party can evade the protocol by radio transmission etc. In the B2B setting, it is realistic that before final deployment of a new controller software (but after the verification), the integrity of the seal is checked. Thus, there is no business incentive for the supplier to break the seal.

The supplier has total control over the information leaving the production site. Thus, it can also prevent attempts by the amanat to leak information by sending messages at specific time points. Because the supplier can read all outgoing messages, there is also a convincing argument for the supplier's non-technical management that no sensitive information is leaking. In our opinion, this simplicity of the amanat protocol is a major advantage for practical application.

Organization of the Paper. In Section 2, we survey related work and discuss alternative approaches to the amanat protocol. The protocol is described in detail in Section 3, and the correctness is addressed in Section 4. The paper is concluded in Section 5.

2   Related Work and Alternative Solutions

The last years have seen renewed activity in the analysis of executables from the verification and programming languages community. Despite remarkable advances (see e.g. [15–18]), the computer-aided analysis of executables remains a hard problem; natural applications are reverse engineering, automatic detection of low level errors such as memory violations, as well as malicious code detection [19, 20]. The technical difficulties in the direct analysis of executables are often exacerbated by code obfuscation to prevent reverse engineering, or, in the case of malware, recognition of the malicious code. Although dynamic analysis [21] and black box testing [22, 23] are relatively immune to obfuscation, they only give a limited assurance of system correctness.

The current paper is orthogonal to executable analysis. We consider a scenario where the software author is willing to assert the quality of the source code by formal methods, but not willing or able to make the source code available to the customer. It is evident that the visibility of the source code to the amanat and the cooperation of the software author/supplier significantly increase the leverage of formal methods.

Proof-Carrying Code [24] is able to generate certificates directly from binaries, but only for a restricted class of safety policies. It is evident that a proof for a non-trivial system property will for all practical purposes explain the internal logic of the binary. Thus, publishing this proof is tantamount to losing intellectual property.

The current paper takes an engineer's view on computer security. The results of the paper are quite specific to verification, as it exploits the conceptual difference between the source code and the executable. While we are aware of advanced methods such as

secure multiparty computation [25] and zero-knowledge proofs [26], we believe that they are not practicable for our problem. To implement secure multiparty computation, it would be necessary to convert significant parts of the model checking tool chain into a Boolean circuit which is not a realistic option. To apply zero-knowledge proofs, one would require the verification tools to produce highly structured and detailed formal proofs. Except for the provers in item 2 of the list in Section 1, it is impractical to obtain such proofs by state of the art technology. More generally, we believe that any advanced method for which secrecy is not intuitively clear to the supplier will be hard to establish in practice. Thus, we are convinced that the conceptual simplicity of our protocol is an asset for practical applicability.

3   The Amanat Protocol

The amanat protocol aims to resolve the conflict between the code customer Cus who wants to verify the source code, and the code supplier Sup who needs to protect its IP. To this end, the amanat Ama computes a certificate which contains enough information to assure the correctness of the program. On the other hand, to secure the IP of Sup, the certificate must not reveal any information beyond the intentionally communicated correctness properties.

3.1   Requirements and Tool Landscape

To make the protocol requirements more precise, we fix some notation and assumptions about the tool landscape. Note that all tools are available to all involved parties.

The *compiler* Compiler takes an input source and computes an executable exec = Compiler(source). Note that Compiler does not take any other input. In practice, this means that source can be thought of as a directory tree containing a make file, and Compiler stands for the tool chain composed of the make command, the compiler, the linker etc.

The *verification tool* Verifier also takes the input source and computes two verification verdicts, $\log_{Sup}$ and $\log_{Cus}$. Here, $\log_{Sup}$ is the "internal" verdict for the supplier which may contain, for example, detailed IP-critical information such as counterexamples or witnesses for certain properties. The second output $\log_{Cus}$ in contrast contains only uncritical verification verdicts about which Sup and Cus have agreed beforehand. Similar as for the compiler, we assume that Verifier does not take any other input parameters. In particular, this means that the specifications are part of source, i.e., they are agreed between the parties and output into $\log_{Cus}$ together with the verification result. Moreover, all auxiliary information necessary for a successful run of Verifier– command line parameters, code annotations, abstraction functions *etc.* – are provided by Sup as part of source.

Before we formally describe the cryptographic primitives for signing and verifying messages, we note that the underlying algorithms are not deterministic but randomized. This randomization is a countermeasure to attacks against naive implementations of RSA and other schemes which exploit algebraically related messages, see for example [27]. In most applications, the randomization is not important for the protocol, as

each participant can locally generate random values. In our protocol however, we have to make sure that the signatures generated by Ama do not contain hidden information for Cus. The way for Ama to leak information to Cus would be to replace the random bits by specifically chosen bits which describe (part of) the source code, similar to steganography [28]. Then, Cus could try to reconstruct the bits from the received message. To exclude this possibility, our protocol will enforce Ama to commit its random bits *before* it sees the source code. Thus, in our description of the cryptographic primitives, we have to treat the random values explicitly.

We also note that in our discussions of randomized algorithms, we usually describe the behavior of the algorithm as it occurs in all but a negligible fraction of the executions of the algorithm [29].

- All parties employ the same *asymmetric encryption and signing scheme* [30] which is based upon RSA [31] and SHA [32]. Given a key pair $\langle K_{pri}, K_{pub} \rangle$ and a message m, we write $c = K_{pub}(m)$ for the encryption of m with key K_{pub} yielding the cipher text c. Similarly, $m = K_{pri}(c)$ denotes the decryption of the cipher text c with key K_{pri} resulting again in the original message m. Furthermore, we write $s = \mathsf{csign}(K_{pri}, m, R)$ for the signature s of a message m signed with key K_{pri} and generated with random seed R. If a signature s is valid and has been generated with seed R, then $\mathsf{cverify}(K_{pub}, m, s, R)$ will succeed and fail otherwise. In situations where the random seed is of no concern, we can also use $\mathsf{cverify}(K_{pub}, m, s)$ which succeeds if s is a valid signature. [3] The algorithms for encryption, decryption, signature generation and signature verification are assumed to require polynomial time with respect to the length of their inputs.
- *Communication Channels.* We assume that the channels between Sup, Cus and Ama are secure, i.e., the protocol is not concerned with eavesdropping on these channels. Moreover, all ingoing and outgoing information for Ama is controlled by Sup, i.e., Sup can manipulate all data exchanged between Ama and Cus.

Having fixed the environment and the notation, we can paraphrase the requirements in a more precise manner:

1. *Conformance* enables Cus to validate that exec and $\log_{\mathsf{Cus}}$ have been produced from the same source.
2. *Secrecy* prevents Cus from extracting, by any tractable process, any IP of Sup except exec and $\log_{\mathsf{Cus}}$.

We note that some of the possible verification tasks discussed in Section 1 – in particular 7, 10, 11, 12 – are concerned with non-functional properties of the source code which do not affect the executable produced by the compiler. The conformance property proves to the customer that at the time of compilation, a source with the required properties did exist. Thus, in the case of a legal conflict, a court can require the supplier to provide a source code which (i) compiles into the purchased executable, and (ii) produces the same verification output $\log_{\mathsf{Cus}}$. There is no mathematical guarantee however, that the revealed code will be *identical* to the original code. This stronger property can be achieved by requiring Verifier to compute a hash of source, and output it into $\log_{\mathsf{Cus}}$.

---

[3] The existence of the 4-parameter variant of $\mathsf{cverify}$ is specific to the chosen scheme [30].

3.2 Summary Description of the Protocol

Our protocol is based on the principle that Cus trusts Ama, and thus, Cus will believe that a verification verdict $\log_{\text{Cus}}$ originating from Ama is conformant with a corresponding binary exec. Therefore, Cus and Sup install Ama at Sup's site such that Sup can use Ama to generate trusted verification verdicts subsequently. On the other hand, Sup controls all the communication to and from Ama and consequently Sup is able to prohibit the communication of any piece of information beyond the verification verdict, i.e., Sup can enforce the *secrecy* of its IP. To ensure that Sup does not alter the verdict of Ama, Ama signs the verdicts with a key which is only known to Ama and Cus but not to Sup. Also, to ensure that the tools Compiler and Verifier given to Ama are untampered, Sup must provide certificates which guarantee that these tools have been approved by Cus.

A protocol based on this simple idea does indeed ensure the conformance property, but a naive implementation with common cryptographic primitives may fail to guarantee the secrecy property: As argued above, the certificates generated by Ama involve random seeds, and Sup *cannot check* that these random seeds do not carry hidden information. In our protocol, to prohibit such hidden transmission of information, Ama is not allowed to generate the required random seeds after it has accessed source. Instead, Ama generates a large supply of random seeds *before* it has access to source, and sends them to Sup. In this way, Ama commits to the random seeds, because later, Sup will check that Ama actually uses the random values which it has sent before. Thus, Ama is not able to encode any information about source into these seeds.

The only remaining problem is that Sup is *not allowed to know the random seeds in advance,* since it could use this knowledge to compromise the cryptographic security of the certificates computed by Ama. Thus, Ama encrypts the random seeds before transmitting them to Sup. Each random seed is encrypted with a specific key, and each time a random seed is used by Ama, the corresponding key is revealed to Sup.

3.3 Detailed Protocol Description

Our protocol consists of three phases, namely the *installation,* the *session initialization,* and the *certification.*

Installation Phase. Cus initializes Ama with a master key pair $\langle K^m_{\text{Cus}}, K^m_{\text{Pub}} \rangle$ which will be used later to exchange a session key pair. Then, Ama is transported to and installed at Sup's site. All further communication between Ama and Cus will be controlled by Sup.

I1 *Master Key Generation*   *[Cus]*
    Cus generates the master keys $\langle K^m_{\text{Cus}}, K^m_{\text{Pub}} \rangle$ and initializes Ama with $\langle K^m_{\text{Cus}}, K^m_{\text{Pub}} \rangle$.
I2 *Installation of the Amanat*   *[Sup, Cus]*
    Ama is installed at Sup's site and Sup receives K^m_{Pub}.

Session Initialization Phase. After installation, Sup and Cus must agree on a specific Verifier and Compiler. Once Verifier and Compiler have been fixed, the session initialization phase starts: First, Cus generates a new pair of session keys $\langle K_{\text{Cus}}, K_{\text{Pub}} \rangle$ and

sends them to Ama via Sup. Then, the new session keys are used to produce certificates $\text{cert}_{\text{Verifier}}$ and $\text{cert}_{\text{Compiler}}$ for Verifier and Compiler, respectively. Sup checks the contents of the certificates and uses them, if they are indeed valid certificates for Verifier and Compiler, to setup Ama with Verifier and Compiler. Ama in turn accepts Verifier and Compiler if their certificates are valid.

In the last step of the initialization, Ama generates a supply of random seeds $R_1, \ldots, R_t$ for t subsequent executions of the certification phase. It also generates a sequence of key pairs $\langle KR^1_{\text{Cus}}, KR^1_{\text{Pub}} \rangle, \ldots, \langle KR^t_{\text{Cus}}, KR^t_{\text{Pub}} \rangle$ for each random seed R_i. Ama finally encrypts each random seed to obtain and send $KR^i_{\text{Pub}}(R_i)$ to Sup. Ama and Sup both keep a variable round which is initialized to 0 and will be incremented by 1 for each execution of the certification phase.

S1 *Session Key Generation*   *[Cus, Sup]*
Cus generates the session keys $\langle K_{\text{Cus}}, K_{\text{Pub}} \rangle$ and sends $K^m_{\text{Pub}}(K_{\text{Cus}})$ and K_{Pub} to Sup. Sup forwards $K^m_{\text{Pub}}(K_{\text{Cus}})$ and K_{Pub} unchanged to Ama.

S2 *Generation of the Tool Certificates*   *[Cus]*
Cus computes the certificates

- $\text{cert}_{\text{Verifier}} = \text{csign}(K_{\text{Cus}}, \text{Verifier})$ and
- $\text{cert}_{\text{Compiler}} = \text{csign}(K_{\text{Cus}}, \text{Compiler})$.

Cus sends both certificates to Sup.

S3 *Supplier Validation of the Tool Certificates*   *[Sup]*
Sup checks the contents of the certificates, i.e., Sup checks that

- $\text{cverify}(K_{\text{Pub}}, \text{Verifier}, \text{cert}_{\text{Verifier}})$ and
- $\text{cverify}(K_{\text{Pub}}, \text{Compiler}, \text{cert}_{\text{Compiler}})$ succeed.

If one of the checks fails, Sup aborts the protocol.

S4 *Amanat Tool Transmission*   *[Sup]*
Sup sends to Ama both Verifier and Compiler as well as the certificates $\text{cert}_{\text{Verifier}}$ and $\text{cert}_{\text{Compiler}}$.

S5 *Amanat Validation of the Tool Certificates*   *[Ama]*
Ama checks whether Verifier and Compiler are properly certified, i.e., it checks whether

- $\text{cverify}(K_{\text{Pub}}, \text{Verifier}, \text{cert}_{\text{Verifier}})$ and
- $\text{cverify}(K_{\text{Pub}}, \text{Compiler}, \text{cert}_{\text{Compiler}})$ succeed.

If this is not the case, then Ama refuses to process any further input.

S6 *Amanat Random Seed Generation*   *[Ama]*
Ama generates

- a series of random seeds $R_1, \ldots, R_t$ together with a series of corresponding key pairs $\langle KR^1_{\text{Cus}}, KR^1_{\text{Pub}} \rangle, \ldots, \langle KR^t_{\text{Cus}}, KR^t_{\text{Pub}} \rangle$,
- encrypts the random seeds with the corresponding keys $KR^i_{\text{Pub}}(R_i)$ for $i = 1, \ldots, t$, and
- initializes round counter round $= 0$.

Ama then sends all $KR^i_{\text{Pub}}(R_i)$ and KR^i_{Pub} for $i = 1, \ldots, t$ to Sup.

Certification Phase. Ama is now ready for the certification phase, i.e., it will accept source and produce a certified verdict on source which can be forwarded to Cus and whose trustworthy origin can be checked by Cus.

During certification, Ama runs Verifier and Compiler on source, generates a certificate cert for the output $\log_{\mathsf{Cus}}$ dedicated to Cus. The certificate is based upon the random seed R_{round} which Ama committed to use in this round of the certification protocol during the session initialization phase. Ama sends the certificate cert, the outputs $\log_{\mathsf{Sup}}$ and $\log_{\mathsf{Cus}}$, and the key $KR_{\mathsf{Cus}}^{\mathrm{round}}$ to Sup.

To validate secrecy, Sup computes the random seed $R_{\mathrm{round}} = KR_{\mathsf{Cus}}^{\mathrm{round}}(KR_{\mathsf{Pub}}(R_{\mathrm{round}}))$ which Ama supposedly used for the generation of cert. Then Sup checks that the certificate cert is indeed a valid certificate and is based upon the random seed R_{round}. If this is the case, i.e., the certificate is valid and is generated based on the predetermined random seed, then Ama cannot hide any unintended information in the certificates. If the checks fails, Sup aborts the protocol. Depending on the output of the Verifier, Sup decides whether to forward the results to Cus or whether to abort the certification phase. Finally, Cus checks conformance of output $\log_{\mathsf{Cus}}$ using cert.

C1 *Source Code Transmission*   *[Sup]*
   Sup sends source to Ama.
C2 *Source Code Verification by the Amanat*   *[Ama]*
   Ama computes
     – the verdict $\langle\log_{\mathsf{Sup}}, \log_{\mathsf{Cus}}\rangle = \mathsf{Verifier}(\mathsf{source})$ of Verifier on source,
     – the binary $\mathsf{exec} = \mathsf{Compiler}(\mathsf{source})$,
     – increments the round counter round, and
     – computes $\mathsf{cert} = \mathsf{csign}(K_{\mathsf{Cus}}, \langle\mathsf{exec}, \log_{\mathsf{Cus}}\rangle, R_{\mathrm{round}})$.
   Ama sends exec, $\log_{\mathsf{Sup}}$, $\log_{\mathsf{Cus}}$, cert, and $KR_{\mathsf{Cus}}^{\mathrm{round}}$ to Sup.
C3 *Secrecy Validation*   *[Sup]*
   Upon receiving exec, $\log_{\mathsf{Sup}}$, $\log_{\mathsf{Cus}}$, cert, and $KR_{\mathsf{Cus}}^{\mathrm{round}}$, Sup
     – decrypts the random seed $R_{\mathrm{round}} = KR_{\mathsf{Cus}}^{\mathrm{round}}(KR_{\mathsf{Pub}}^{\mathrm{round}}(R_{\mathrm{round}}))$, and
     – verifies that $\mathsf{cverify}(K_{\mathsf{Pub}}, \langle\mathsf{exec}, \log_{\mathsf{Cus}}\rangle, \mathsf{cert}, R_{\mathrm{round}})$ succeeds.
   If the checks fails, Sup **concludes that the secrecy requirement was violated**, and refuses to further work with Ama.
   Otherwise, Sup evaluates $\log_{\mathsf{Cus}}$ and $\log_{\mathsf{Sup}}$ and decides whether to deliver the binary exec, $\log_{\mathsf{Cus}}$, and cert to Cus in step **C4** or whether to abort the protocol.
C4 *Conformance Validation*   *[Cus]*
   Upon receiving exec, $\log_{\mathsf{Cus}}$, and cert, Cus verifies that $\mathsf{cverify}(K_{\mathsf{Pub}}, \langle\mathsf{exec}, \log_{\mathsf{Cus}}\rangle, \mathsf{cert})$ succeeds.
   If the checks fails, Cus **concludes that the conformance requirement was violated**, and refuses to further work with Sup.
   Otherwise Cus evaluates the contents of $\log_{\mathsf{Cus}}$ and decides whether the verification verdict supports the purchase of the product exec.

4 Protocol Correctness

In this section, we prove conformance and secrecy of our protocol using standard cryptographic assumptions. Following [14], we assume that the public-key encryption is *semantically secure* and that the used signature scheme is *secure against adaptive chosen*

message attacks, such as the RSA-based scheme proposed in [30]. We briefly introduce these security properties:

Semantic security means that whatever can be learnt from the ciphertext within probabilistic polynomial time, can be computed, again within probabilistic polynomial time, from the length of the plaintext alone. Formally, semantic security means that each probabilistic polynomial time algorithm which takes as arguments a security parameter, a public key, a number of messages encrypted with this key, the respective messages lengths, and any further partial information on the messages, can be replaced by another probabilistic polynomial time algorithm which only receives the security parameter, the message lengths, and the partial information on the messages [14]. In other words, no probabilistic polynomial time algorithm can extract any information from a set of encrypted messages.

An *adaptive chosen message attack* is an attack against a signature scheme, where the attacker has access to an oracle which can sign arbitrary messages, and uses this ability to sign some new message *without consulting the oracle.* More formally, a *signing oracle* $S[K_{\mathsf{Cus}}]$ with private key K_{Cus} is a function which takes a message m and returns a signature $s = \mathsf{csign}(K_{\mathsf{Cus}}, m, R)$ for a uniformly and randomly chosen random seed R. An attack is a forging algorithm F which (i) knows the public key K_{Pub} and (ii) has access to the signing oracle $S[K_{\mathsf{Cus}}]$, where K_{Cus} is the private key corresponding to K_{Pub}. The algorithm F is allowed to query $S[K_{\mathsf{Cus}}]$ for an arbitrary number of signatures. F can adaptively choose the messages to be signed, i.e., each newly chosen message can depend on the outcome of the previous queries. At the end of the computation, a successful attack F must output a message m and a signature s such that $\mathsf{cverify}(K_{\mathsf{Pub}}, m, s)$ succeeds, although m has never been sent to $S[K_{\mathsf{Cus}}]$. A signature scheme is secure against adaptive chosen message attacks, if there is no probabilistic polynomial time algorithm F which has a non-negligible success probability.

We can now precisely state the main theorems.

Theorem 1 (Conformance). *If the protocol terminates (in Step **C4** of the certification phase) with the customer* Cus *accepting the binary* exec *and the output file* $\log_{\mathsf{Cus}}$, *then* exec *and* $\log_{\mathsf{Cus}}$ *must be produced from the same* source *in all but a negligible fraction of the protocol executions (under standard cryptographic assumptions).*

Proof Sketch. Towards a contradiction, we assume that with non-negligible probability, Sup can forge a certificate which is accepted by Cus in step **C4**. Thus, Sup computes a certificate cert for a pair $\langle \mathsf{exec}, \log_{\mathsf{Cus}} \rangle$ which has not been signed by Ama but is accepted by Cus. Using semantic security, we show that such a malicious instance MSup of Sup gives rise to a forging algorithm F which implements a successful adaptive chosen message attack. This implies that the underlying signature scheme is not secure against adaptive chosen message attacks—which is a contradiction. □

We present a more extensive proof of Theorem 1 in [33]. We now turn to secrecy, which, not surprisingly, is quite straight forward to prove.

Theorem 2 (Secrecy). *By the execution of the protocol,* Cus *cannot extract any piece of information on the source* source *which is not contained in* exec *and* $\log_{\mathsf{Cus}}$.

Proof. During the execution of the protocol, Cus receives the binary exec, the output file $\log_{\mathsf{Cus}}$, and the certificate cert. The certificate $\mathsf{cert} = \mathsf{csign}(K_{\mathsf{Cus}}, \langle \mathsf{exec}, \log_{\mathsf{Cus}} \rangle, \mathsf{R_i})$ can be generated from exec, $\log_{\mathsf{Cus}}$, the key K_{Cus}, and the underlying random seed R_i. Cus generates K_{Cus} itself and obtains access to exec and to $\log_{\mathsf{Cus}}$. Thus the only additional information communicated from Ama to Sup is the underlying random seed R_i. But this random seed R_i has been fixed by Ama before having access to source, and consequently Ama cannot encode any information on the source source which is not contained in exec and $\log_{\mathsf{Cus}}$ into the certificate. $\qquad\square$

5 Conclusion

We have introduced the amanat protocol which facilitates software verification without violating IP rights on the source code. The intended scenario for our protocol is a B2B setting with a small numbers of customers, e.g. controller software and device drivers.

We also envision wider applications of our protocol in a B2C setting, i.e., for commercial-off-the-shelf software. In this case, the customer party of the amanat protocol will not be enacted by an end customer, but by a certification agency which provides commercial verification services. A detailed exploration of this scenario will be part of future work.

References

1. Heinecke, H.: Automotive Open System Architecture-An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures. In: Society of Automotive Engineers World Congress. (2004)
2. Broy, M.: Challenges in automotive software engineering. In: ICSE '06. (2006) 33–42
3. Ball, T., Cook, B., Levin, V., Rajamani, S.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Proc. of IFM. (2004) Invited talk.
4. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: SPIN Workshop on Model Checking of Software. Volume 2057 of LNCS. (2001)
5. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proc. 29th POPL, Association for Computing Machinery (2002) 58–70
6. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In: Proc. ICSE '03. (2003) 385–395
7. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: CAV. (2006) 415–418
8. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: SAS. (2006) 240–260
9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astrée analyser. In: ESOP 2005. Volume 3444 of LNCS. (2005) 21–30
10. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3) (2002) 217–298

11. Wiedijk, F., ed.: The Seventeen Provers of the World. Volume 3600 of Lecture Notes in Computer Science. (2006)
12. Wenzel, I., Kirner, R., Rieder, B., Puschner, P.P.: Measurement-based worst-case execution time analysis. In: SEUS 2005. (2005) 7–10
13. Ferdinand, C., Heckmann, R., Wilhelm, R.: Analyzing the worst-case execution time by abstract interpretation of executable code. In: ASWSD 2004. (2004) 1–14
14. Goldreich, O.: Foundations of Cryptography. Volume II: Basic Applications. Cambridge University Press (2004)
15. Balakrishnan, G., Reps, T.: DIVINE: DIscovering Variables IN Executables. In: Proc. VM-CAI '07. Volume 4349 of LNCS. (2007) 1–28
16. Debray, S.K., Muth, R., Weippert, M.: Alias analysis of executable code. In: Proc. 26th POPL. (1999)
17. Reps, T.W., Balakrishnan, G., Lim, J., Teitelbaum, T.: A next-generation platform for analyzing executables. In: Third Asian Symposium on Programming Languages and Systems (APLAS 2005). Volume 3780 of LNCS. (2005) 212–229
18. Cifuentes, C., Fraboulet, A.: Intraprocedural static slicing of binary executables. In: ICSM. (1997) 188–195
19. Christodorescu, M., Jha, S., Seshia, S.A., Song, D.X., Bryant, R.E.: Semantics-aware malware detection. In: IEEE Symposium on Security and Privacy. (2005) 32–46
20. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: DIMVA. (2005) 174–187
21. Colin, S., Mariani, L.: Run-Time Verification. In: Model-based Testing of Reactive Systems. Volume 3472 of Lecture Notes in Computer Science. Springer (2005)
22. Lee, D., Yannakakis, M.: Testing finite-state machines: State identification and verification. IEEE Transactions on Computers **43**(3) (1994) 306–320
23. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines – a survey. Proceedings of the IEEE **84**(8) (1996) 1090–1126
24. Necula, G.C.: Proof-Carrying Code. In: Proc. 24th POPL, Paris, France, January 15–17, 1997. New York, NY, Association for Computing Machinery (1997) 106–119
25. Goldreich, O.: Secure multi-party computation. Final Draft, Version 1.4 (2002)
26. Ben-Or, M., Goldreich, O., Goldwasser, S., Hastad, J., Kilian, J., Micali, S., Rogaway, P.: Everything Provable is Provable in Zero-Knowledge. In: CRYPTO '88. (1988) 37–56
27. Dolev, D., Dwork, C., Naor, M.: Non-Malleable Cryptography. Siam Journal on Computing **30**(2) (2000) 391–437
28. Petitcolas, F., Katzenbeisser, S., eds.: Information Hiding Techniques for Steganography and Digital Watermarking. Artech House (2000)
29. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1997)
30. Cramer, R., Shoup, V.: Signature Schemes Based on the String RSA Assumption. ACM Transactions on Information and System Security **3**(3) (2000) 161–185
31. Rivest, R.L., Shamir, A., Adleman, L.M.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM **21**(2) (1978) 120–126
32. NIST: NIST FIPS PUB 180-1, Secure Hash Standard (1995)
33. Chaki, S., Schallhart, C., Veith, H.: Verification Across Intellectual Property Boundaries. http://arxiv.org/abs/cs.OH/0701187 (2007)