Verification Across Intellectual Property Boundaries

SAGAR CHAKI, Software Engineering Institute
CHRISTIAN SCHALLHART, Oxford University
HELMUT VEITH, Vienna University of Technology

In many industries, the importance 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 an 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.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Validation; D.2.9 [**Management**]: Software Quality Assurance (SQA); K.6.5 [**Security and Protection**]: Authentication

General Terms: Verification, Management, Security

Additional Key Words and Phrases: Intellectual Property, Supply-Chain

ACM Reference Format:
Chaki, S., Schallhart, C., and Veith, H. 2012. Verification Across Intellectual Property Boundaries. ACM Trans. Softw. Eng. Methodol. 9, 4, Article 39 (March 2012), 12 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

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

Supported by the European FP6 project ECRYPT (IST-2002-507932), the DFG research grant FORTAS (VE 455/1-1), and the Predictable Assembly from Certifiable Components (PACC) initiative at the Software Engineering Institute, Pittsburgh, USA.
Authors' addresses: S. Chaki, Software Engineering Institute; C. Schallhart, Computer Science Department, Oxford University; H. Veith, Formal Methods in Systems Engineering Group, Vienna University of Technology.
A preliminary version of this paper was presented at CAV 2007 [Chaki et al. 2007a] in Berlin, Germany, and a version with all proofs can be found as CoRR report [Chaki et al. 2007b].

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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 [Heinecke 2004; Broy 2006] and Windows device drivers [Ball et al. 2004]. Although the suppliers may use verification 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.

This setting constitutes a principal conflict between the *supplier* Sup who owns the source code, and the *customer* Cus who purchases *only* 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 ensure that no information about the source code besides 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 satisfies both 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 supports 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 leave the supplier again.

It is evident that clauses (i) and (ii) make it quite infeasible to find human amanats; instead, our protocol utilizes 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, Ama produces a cryptographic certificate which proves that the purchased executable *has been derived by the amanat from the same source code as the verification verdict.*

To achieve this goal, we employ 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. Hence, the protocol satisfies clauses (i) and (ii).

Figure 1 presents a high-level illustration of the protocol: If the code supplier Sup and the customer Cus utilize the amanat protocol, they first install an amanat server Ama such that (a) Cus is assured that Sup is unable to tamper with the amanat, and (b) Sup gains complete control over the communication link between Ama and Cus. The customer Cus equips Ama with a public/private key pair, such that Ama can authenticate its messages to be delivered to Cus. While the public key is handed to Sup, the private key is kept secret from Sup. Then, once provided with the tools Compiler

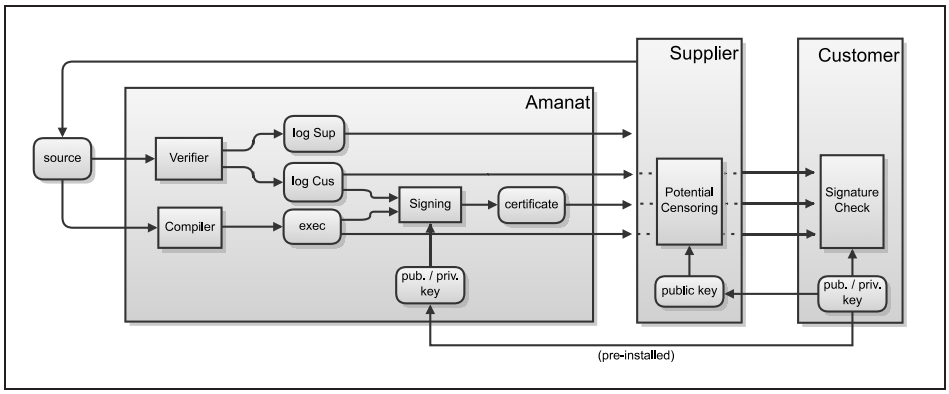

Fig. 1. A High-Level View of the Amanat Protocol

and Verifier, Ama is ready to compute certified verification verdicts: Sup assembles the sources source and sends them to Ama. Ama runs Verifier to obtain the outputs $\log_{Sup}$ and $\log_{Cus}$, dedicated to Sup and Cus, respectively: $\log_{Sup}$ may contain IP-critical but development-relevant information, while $\log_{Cus}$ is only allowed to contain the verification verdict. Parallel to the verification, Ama uses Compiler to compile the binary exec. Using its private key, Ama computes a certificate cert which authenticates exec together with $\log_{Cus}$ as being computed from the same source. All these results are returned to Sup. Then Sup checks whether all computations have been performed as expected, i.e., whether exec and $\log_{Cus}$ resulted from respective invocations of Compiler and Verifier, and whether cert has been computed properly. Since the computation of cert involves random bits, the amanat protocol allows Sup to check that Ama chose these random bits *before* accessing source—such that they cannot contain any information on source. If these checks succeed, Sup is assured that the secrecy of its IP has been preserved. After evaluating $\log_{Sup}$ and $\log_{Cus}$ regarding its content, Sup decides whether to forward exec, $\log_{Cus}$, and cert to Cus across the IP boundary or not. If the results are forwarded to Cus, cert is checked by Cus and if the certificate is valid, Cus accepts exec and $\log_{Cus}$ as being conformant.

Verification by Model Checking and Beyond. Motivated by discussions with industrial collaborators, we primarily intended our protocol to facilitate software model checking across IP boundaries in a B2B setting where suppliers and customers are businesses. Our guiding examples for this 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 [Ball and Rajamani 2001; Henzinger et al. 2002; Chaki et al. 2003; Clarke et al. 2004; Cook et al. 2006; Gotsman et al. 2006; Podelski and Rybalchenko 2007].

At the same time, intensive software development have conquered traditional industries which rely on deep supply chains, such as the automotive industries: A number of initiatives have been started to establish standardized and industry-wide accepted computing environments, most notably AUTOSAR [07: 2007a], OSEK [07: 2007c], or JASPAR [07: 2007b]. These standardization efforts originate in the need to *integrate software components developed by various companies along a deep supply chain,* demanding a component-wise verification in presence of heavy IP interests [Pretschner et al. 2007].

The amanat protocol provides a general framework to perform source dependent validation tasks in presence of IP boundaries and is therefore *not restricted* to pure

verification: For example, it may be necessary for customers and suppliers to communicate some software design details without revealing the underlying source code. In this case, 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. This is possible, because the amanat can run *any* verification/validation tool whose output does not compromise the secrecy of the source code, including e.g. static analysis tools such as ASTREE [Cousot et al. 2005] or TVLA [Sagiv et al. 2002], or theorem provers for checking correctness proofs, such as PVS, ISABELLE, Coq or any other prover [Wiedijk 2006].

We note that in all these scenarios the code supplier *bears the burden of proof*: Ama is provided with source and must be able to verify with Verifier alone that source and the resulting binary exec are satisfying the specified requirements. To this end, source may contain auxiliary information supporting Verifier in computing the verification verdict, e.g. a proof to be verfied by a theorem prover to show the conformance of source with its specification. However, this auxiliary information must not influence the verdict but only help computing it.

Security of the Amanat Protocol. We discuss in Section 4 the *secrecy* and *conformance* of the amanat verification protocol. Stronger than term-based proofs in the Dolev-Yao model [Dolev and Yao 1981], the security properties of the protocol assure that under standard cryptographic assumptions, randomized polynomial time attacks against the protocol (involving e.g. guessing the private keys) can succeed only with negligible probability.

The cryptographic protocols require Cus and Ama to *share a secret* unknown to Sup, namely the private key of the customer; this secret enables Ama to authenticate its verdict to Cus and by computing the certificate. Consequently, we need to assure a physical system configuration where Ama can neither be reverse-engineered, nor closely monitored by the supplier. On the other hand, from the point of view of Sup, Ama is an untrusted black box, and thus Sup requires ownership of Ama to ensure it will never return to the customer. Hence, Ama is physically located at the supplier, but in a sealed location or box whose integrity is assured through, e.g., regular checks by the customer, a third party, or sealed hardware [Ravi et al. 2004]. All communication channels of Ama are hardwired to the communication filter of Sup. In B2B settings, such a scenario is practically feasible, as it only requires that the seal is checked after verification and before deployment.

The supplier has total control over the information leaving the production site, and *can read, delay, drop, and modify all outgoing messages*—which is a convincing and easy to explain argument 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. Afterwards in Section 3, we introduce the relevant tools and cryptographic primitives, followed by a brief protocol overview, and a detailed description of the protocol. The secrecy and conformance of the protocol are discussed in Section 4, and the paper is concluded with 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. [Balakrishnan and Reps 2007; Debray et al. 1999; Reps et al. 2005; Cifuentes and Fraboulet 1997; Kinder and Veith 2010]), the computer-aided analysis of executables remains a hard problem. Although dynamic analysis [Colin and Mariani 2005] and

approaches dealing with black box systems [Lee and Yannakakis 1994; 1996; Peled et al. 1999] are relatively immune to obfuscation, they are limited either in the range of systems they can deal with or in the correctness properties they can assure.

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 does not or provide the source code to the customer.

While proof-Carrying Code [Necula 1997] is able to generate certificates for binaries, it is only applicable for a restricted class of safety policies. More importantly, a proof for non-trivial system properties will explain—for all practical purposes—the internal logic of the binary, and thus, publishing this proof is tantamount to losing intellectual property.

The current paper takes an engineer's view on computer security, as it exploits the conceptual difference between source code and executable. While we are aware of advanced methods such as secure multiparty computation [Goldreich 2002] and zero-knowledge proofs [Ben-Or et al. 1988], we believe that they are impracticable for our problem, as such methods cannot be easily wrapped over given validation tools. Finally, we believe that any advanced method without an intuitive proof for its secrecy will be heavily opposed by the supplier—and might therefore 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 resolves the conflict between the customer Cus who wants to verify the source code, and the supplier Sup who needs to protect its IP. To this end, the amanat Ama computes a certificate which contains a verdict on the program correctness, but does not reveal any information beyond the verdict itself.

3.1. Requirements and Tool Landscape

We fix some notation and assumptions about the tool landscape. Our tools are restricted to run in deterministic polynomial time, however, to deal with higher runtime complexities, we pad source, i.e., add a string long enough to upper-bound the runtime of all tools with a polynomial in the padded length of source. By adding random seeds to source, we can also integrate randomized tools into our framework.

Definition 3.1 (*Compiler*). The *compiler* Compiler translates an input source into an executable $\mathsf{exec} = \mathsf{Compiler}(\mathsf{source})$ in deterministic polynomial time.

Note that Compiler does not take any further input. In practice, this means that source is a directory tree and that Compiler is a tool chain composed of compiler, linker etc.

Definition 3.2 (*Verification Tool*). The *verification tool* Verifier takes the input source and computes in deterministic polynomial time two verification verdicts, $\mathsf{log}_{\mathsf{Sup}}$ and $\mathsf{log}_{\mathsf{Cus}}$, i.e., $\langle \mathsf{log}_{\mathsf{Sup}}, \mathsf{log}_{\mathsf{Cus}} \rangle = \mathsf{Verifier}(\mathsf{source})$.

Here, $\mathsf{log}_{\mathsf{Sup}}$ is a detailed verdict for the supplier possibly containing IP-critical information such as counterexamples or witnesses for certain properties. The second output $\mathsf{log}_{\mathsf{Cus}}$ in contrast contains only uncritical verification verdicts which Sup and Cus have agreed upon beforehand. Similar as for the compiler, we assume that Verifier does not take any inputs besides source. Hence source contains auxiliary information, such as command line parameters, and the specification to be checked—allowing Ama to output the verification results together with their specifications into $\mathsf{log}_{\mathsf{Cus}}$ to be delivered to Cus.

As in the case of Compiler, Verifier is not restricted to consist of a single tool. On the contrary, Verifier can comprise a whole set of verification tools, as long as they are tied together to produce a single pair $\langle \log_{\mathsf{Sup}}, \log_{\mathsf{Cus}} \rangle$ of combined outputs.

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

Definition 3.3 (*Conformance*). An execution of the amanat protocol is *conformant,* if the delivered binary exec and verdict $\log_{\mathsf{Cus}}$ have been produced from the same source.

Definition 3.4 (*Secrecy*). An execution of the amanat protocol ensures *secrecy*, if all information provided to Cus in the course of the protocol is either directly contained in or implied by exec and $\log_{\mathsf{Cus}}$.

The goal of the Amanat protocol is to give mathematical guarantees for these two properties for all (but a negligible fraction) of protocol executions—based on two assumptions: First, the communication channels between Sup, Cus, and Ama must be secure, i.e., the protocol is not designed to cope with orthogonal risks such as eavesdropping or malicious manipulations on these channels. Second, we assume that all ingoing and outgoing information for Ama is controlled by Sup, i.e., Sup can manipulate all data exchanged between Ama and Cus.

3.2. Cryptographic Primitives

Before we formally describe the primitives for encrypting, decrypting, 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 [Dolev et al. 2000]. In many protocols, the randomization can be treated as technical detail, as each participant can locally generate random values. But in our protocol, we must ensure that the signatures generated by Ama do not contain hidden information for Cus—and must hence deal with randomization explicitly: Using methods from steganography [Petitcolas and Katzenbeisser 2000], Ama could encode source code properties into allegedly randomly generated bits. To preclude this possibility, our protocol forces Ama to commit its random bits *before* it sees the source code.

Below, we define schemes for encrypting and signing messages. In case of the signature scheme, we also add procedures with explicit randomization parameters. As both schemes use an asymmetric key-pair, we assume that the same pair $\langle k_{priv}, k_{pub} \rangle$ is applicable for both.

Definition 3.5 (*Public-Key Encryption Scheme*). Given a *key pair* $\langle k_{priv}, k_{pub} \rangle$, we define the encryption and decryption and their respectively required computational complexity bound (with respect to length of their inputs and security parameters) as follows:

— **Encryption:** For a *plaintext* message m, we write $c = k_{pub}(m)$ to denote the *encryption* of m with key k_{pub} yielding the *ciphertext* c (probabilistic polynomial time).
— **Decryption:** Similarly, $m = k_{priv}(c)$ denotes the *decryption* of the ciphertext c with key k_{priv} resulting again in the original message m (deterministic polynomial time).

Definition 3.6 (*Public-Key Signature Scheme*). Given a *key pair* $\langle k_{priv}, k_{pub} \rangle$, we define the following operations running in deterministic polynomial time (with respect to length of their inputs and the security parameter):

— **Signature Generation:** We write $s = \mathsf{csign}(k_{priv}, m, R)$ for the *signature* s of a message m signed with key k_{priv} and generated with random seed R.

— **Signature Verification:** $\mathsf{cverify}(k_{pub}, m, s)$ denotes the verification result of a signature s for message m with key k_{priv}. The verification succeeds, iff there exists a random seed R such that $s = \mathsf{csign}(k_{priv}, m, R)$ holds.
— **Random Seed Extraction:** We write $R = \mathsf{cextract}(s)$ for $s = \mathsf{csign}(k_{priv}, m, R)$ to extract the random seed R used in a signature s generated for message m with key k_{priv}.
— **Signature Verification with Fixed Random Seed:** We write $\mathsf{cverify}(k_{pub}, m, s, R)$ to check whether a signature s for message m and k_{priv} has been generated with random seed R, i.e., $\mathsf{cverify}(k_{pub}, m, s, R)$ succeeds iff $s = \mathsf{csign}(k_{priv}, m, R)$ holds.

Thus, besides standard signature generation and verification with $\mathsf{csign}(k_{priv}, m, R)$ and $\mathsf{cverify}(k_{pub}, m, s)$, respectively, we require the existence of two additional procedures: The first one, $\mathsf{cextract}(s)$ extracts the random seed R used in signature s, and the second one, $\mathsf{cverify}(k_{pub}, m, s, R)$ verifies that signature s has been generated with seed R.

Aside providing these interfaces, suitable cryptographic primitives must satisfy the relevant security properties (see Theorem 4.2): In case of the encryption scheme, our requirements are fairly standard and are satisfied by a number of encryption schemes, e.g. one can use ElGamal encryption [ElGamal 1985]. For the signature scheme, we propose to use [Cramer and Shoup 2000] which is based upon RSA [Rivest et al. 1978] and SHA [NIST 1995] and allows to implement all operations described above.

3.3. Summary Description of the Protocol

Our protocol is based on the principle that Cus trusts Ama, and thus, Cus believes that a verification verdict $\mathsf{log}_{\mathsf{Cus}}$ 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. At the same time, 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 ensure the conformance property, but a naive implementation with common cryptographic primitives fails to guarantee secrecy: As argued above, the certificates generated by Ama involve random seeds, and Sup *cannot check* these random seeds for 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. Later, Sup checks that Ama used exactly these random values. 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. Therefore, Ama encrypts each random seed with a specific key before transmitting them to Sup, and reveals the corresponding key when it uses one of its seeds.

3.4. Detailed Protocol Description

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

Installation Phase: After Ama is initialized and installing at the designated site, all communication between Ama and Cus is controlled by Sup.

I1 ***Master Key Generation*** [Cus]
Cus generates the master keys $\langle km_{priv}, km_{pub}\rangle$ and initializes Ama with them.

I2 ***Installation of the Amanat*** [Sup, Cus]
Ama is installed at Sup's site and Sup receives km_{pub}.

Session Initialization Phase: Once Sup and Cus must agree on a specific Verifier and Compiler, the session initialization phase starts: Cus generates a session key pair, sends it in an encrypted manner to Ama via Sup (**S1**), and Verifier and Compiler are certified and handed to Ama (**S2-5**). Next, Ama generates a supply of random seeds $R_1, \ldots, R_t$ for t subsequent executions of the certification phase, along with corresponding key pairs $\langle kr^t_{priv}, kr^t_{pub}\rangle$. Ama encrypts each random seed and sends $kr^i_{pub}(R_i)$ to Sup. Ama and Sup both maintain a variable round which is initialized to 0 and incremented by 1 for each execution of the certification phase (**S6**).

S1 ***Session Key Generation*** [Cus, Sup]
Cus generates the session keys $\langle ks_{priv}, ks_{pub}\rangle$ and sends $km_{pub}(ks_{priv})$ and ks_{pub} to Sup. Sup forwards $km_{pub}(ks_{priv})$ and ks_{pub} unchanged to Ama.

S2 ***Generation of the Tool Certificates*** [Cus]
Cus computes the certificates
— $\mathsf{cert}_{\mathsf{Verifier}} = \mathsf{csign}(ks_{priv}, \mathsf{Verifier})$ and
— $\mathsf{cert}_{\mathsf{Compiler}} = \mathsf{csign}(ks_{priv}, \mathsf{Compiler})$.
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
— $\mathsf{cverify}(ks_{pub}, \mathsf{Verifier}, \mathsf{cert}_{\mathsf{Verifier}})$ and
— $\mathsf{cverify}(ks_{pub}, \mathsf{Compiler}, \mathsf{cert}_{\mathsf{Compiler}})$ succeed.
If one of the checks fails, Sup aborts the protocol.

S4 ***Amanat Tool Transmission*** [Sup]
Sup sends Verifier, Compiler, and the certificates $\mathsf{cert}_{\mathsf{Verifier}}$ and $\mathsf{cert}_{\mathsf{Compiler}}$ to Ama.

S5 ***Amanat Validation of the Tool Certificates*** [Ama]
Ama checks whether Verifier and Compiler are properly certified, i.e., it checks whether
— $\mathsf{cverify}(ks_{pub}, \mathsf{Verifier}, \mathsf{cert}_{\mathsf{Verifier}})$ and
— $\mathsf{cverify}(ks_{pub}, \mathsf{Compiler}, \mathsf{cert}_{\mathsf{Compiler}})$ succeed.
If this check fails, 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 $\langle kr^1_{priv}, kr^1_{pub}\rangle, \ldots, \langle kr^t_{priv}, kr^t_{pub}\rangle$,
— encrypts random seeds with the corresponding keys $kr^i_{pub}(R_i)$ for $i = 1, \ldots, t$, and
— initializes round counter $\mathsf{round} = 0$.
Ama sends $kr^i_{pub}(R_i)$ 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 to produce a certified verdict on source which can be forwarded to Cus and

whose trustworthy origin can be checked by Cus. For certification, Ama runs Verifier and Compiler on source and generates a certificate cert for the binary exec and the verification output $\log_{\mathsf{Cus}}$ dedicated to Cus, all based on the random seed R_{round} (**C1-2**). Upon receiving the certification results along with the key $kr_{priv}^{\mathsf{round}}$, Sup reconstructs the random seed $R_{\mathsf{round}} = kr_{priv}^{\mathsf{round}}(kr_{pub}(R_{\mathsf{round}}))$ which Ama supposedly used for the generation of the certificate. Then, if the certificate is valid and based upon R_{round}—and if the validation verdict is good enough—Sup forwards the certified results to Cus (**C3**). In turn, Cus checks that the certificate is valid and evaluates the validation verdict (**C4**).

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
— $\mathsf{cert} = \mathsf{csign}(ks_{priv}, \langle \mathsf{exec}, \log_{\mathsf{Cus}} \rangle, \mathsf{R}_{\mathsf{round}})$.
Ama sends exec, $\log_{\mathsf{Sup}}$, $\log_{\mathsf{Cus}}$, cert, and $kr_{priv}^{\mathsf{round}}$ to Sup.

C3 ***Secrecy Validation*** [Sup]
Upon receiving exec, $\log_{\mathsf{Sup}}$, $\log_{\mathsf{Cus}}$, cert, and $kr_{priv}^{\mathsf{round}}$, Sup
— decrypts the random seed with $R_{\mathsf{round}} = kr_{priv}^{\mathsf{round}}(kr_{pub}^{\mathsf{round}}(R_{\mathsf{round}}))$,
— checks whether $\mathsf{exec} = \mathsf{Compiler}(\mathsf{source})$ and $\langle \log_{\mathsf{Sup}}, \log_{\mathsf{Cus}} \rangle = \mathsf{Verifier}(\mathsf{source})$ hold, and
— verifies that $\mathsf{cverify}(ks_{pub}, \langle \mathsf{exec}, \log_{\mathsf{Cus}} \rangle, \mathsf{cert}, \mathsf{R}_{\mathsf{round}})$ succeeds.
If the checks fails, Sup **concludes that the secrecy requirement has been violated**, and refuses to proceed with the protocol.
Otherwise, Sup evaluates $\log_{\mathsf{Cus}}$ and $\log_{\mathsf{Sup}}$ and decides whether to deliver exec, $\log_{\mathsf{Cus}}$, and cert to Cus in step **C4** or whether to abort the protocol.

C4 ***Conformance Validation*** [Cus]
Having received exec, $\log_{\mathsf{Cus}}$, and cert, Cus verifies that certificate is valid, i.e., that $\mathsf{cverify}(ks_{pub}, \langle \mathsf{exec}, \log_{\mathsf{Cus}} \rangle, \mathsf{cert})$ succeeds.
If the checks fails, Cus **concludes that the conformance requirement has been violated**, and refuses to proceed with the protocol.
Otherwise Cus evaluates the contents of $\log_{\mathsf{Cus}}$ and decides whether the verification verdict supports the purchase of the product exec.

4. PROTOCOL SECURITY

We designed the amanat protocol aiming at a simple and intuitive argument for its secrecy. Such a straightforward proof is a prerequisite to convince a code supplying company that the protocol keeps their highly valued IP assets safe.

The idea behind this proof is straightforward and does not rely on cryptographic assumptions: As the certificate cert is the only place to transmit additional information from Ama to Cus, we make sure that cert can be computed without *knowing the source itself.* Hence, no information on the source can be possibly hidden in cert.

THEOREM 4.1 (SECRECY). *The amanat protocol enforces secrecy (see Definition 3.4) in all its executions unconditionally.*

More specifically, Cus is unable to extract any piece of information on the source source which is not contained in exec and $\log_{\mathsf{Cus}}$, because the certificate is entirely determined by source and a *pre-committed* random seed.

We state the conformance of our protocol using standard cryptographic assumptions: Following [Goldreich 2004], we assume that the public-key encryption is *semantically secure* and that the signature scheme is *secure against adaptive chosen message attacks*, such as the RSA-based scheme proposed in [Cramer and Shoup 2000].

THEOREM 4.2 (CONFORMANCE). *The conformance property holds in all but a negligible fraction of the accepting protocol executions, under the assumption that (a) the underlying encryption is semantically secure [Goldreich 2004], (b) the signature scheme is secure against adaptively chosen message attacks [Cramer and Shoup 2000], and (c) the supplier* Sup *runs in probabilistic polynomial time.*

Proofs for both theorems can be found in [Chaki et al. 2007b].

5. CONCLUSION

IP boundaries impose an obstacle for the dissemination and application of verification techniques. In the common research scenarios on verfication, a relationship of mutual trust between the software author and the verification engineer is assumed: First, the verification engineer believes that the sources provided by the author have yielded the final binary, and second, the author expects the engineer to respect its IP rights on the provided sources. But in an industrial context, a mutual trust relationship is insufficient as protection against misconduct. We identified two security properties which facilitate verification across IP boundaries: First, *conformance* requires that the verification verdict and the delivered binary are produced from the same source, and second, *secrecy* requires that the customer does not learn anything about the sources except for the information in the binary and the verdict.

Taking the need for conformance and secrecy as starting point, we introduce the *amanat protocol* to satisfy both requirements: The amanat protocol guarantees secrecy unconditionally and conformance based on commonly relied-upon security assumptions.

ACKNOWLEDGMENTS

We are thankful to Josh Berdine and Byron Cook for discussions on the device driver scenario and to Andreas Holzer and Stefan Kugele for comments on early drafts of the paper.

REFERENCES

2007a. Automotive Open System Architecture (AUTOSAR) Consortium. `www.autosar.org`.

2007b. Japan Automotive Software Platform Architecture (JASPAR). `www.jaspar.jp`.

2007c. Offene Systeme und deren Schnittstellen fr die Elektronik in Kraftfahrzeugen. `www.osek-vdx.org`. (eng. name.: Open Systems and the Corresponding Interfaces for Automotive Electronics).

BALAKRISHNAN, G. AND REPS, T. 2007. DIVINE: DIscovering Variables IN Executables. In *Proc. 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '07)*. 1–28.

BALL, T., COOK, B., LEVIN, V., AND RAJAMANI, S. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Proc. 4th International Conference on Integrated Formal Methods (IFM '04)*. 1–20. Invited.

BALL, T. AND RAJAMANI, S. K. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *Proc. 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*. 103–122.

BEN-OR, M., GOLDREICH, O., GOLDWASSER, S., HASTAD, J., KILIAN, J., MICALI, S., AND ROGAWAY, P. 1988. Everything Provable is Provable in Zero-Knowledge. In *Proc. 8th Annual International Cryptology Conference (CRYPTO '88)*. 37–56.

BROY, M. 2006. Challenges in automotive software engineering. In *Proc. 28th International Conference on Software Engineering (ICSE '06)*. 33–42.

CHAKI, S., CLARKE, E., GROCE, A., JHA, S., AND VEITH, H. 2003. Modular verification of software components in C. In *Proc. 25th International Conference on Software Engineering (ICSE '03)*. 385–395.

CHAKI, S., SCHALLHART, C., AND VEITH, H. 2007a. Verification Across Intellectual Property Boundaries. In *Proc. 19th International Conference on Computer Aided Verification (CAV '07)*. 82–94.

CHAKI, S., SCHALLHART, C., AND VEITH, H. 2007b. Verification across intellectual property boundaries. *CoRR abs/cs/0701187*.

CIFUENTES, C. AND FRABOULET, A. 1997. Intraprocedural static slicing of binary executables. In *Proc. 13th International Conference on Software Maintenance (ICSM '97)*. 188–195.

CLARKE, E. M., KROENING, D., AND LERDA, F. 2004. A Tool for Checking ANSI-C Programs. In *TACAS*. 168–176.

COLIN, S. AND MARIANI, L. 2005. *Model-based Testing of Reactive Systems*. Lecture Notes in Computer Science Series, vol. 3472. Springer, Chapter Run-Time Verification.

COOK, B., PODELSKI, A., AND RYBALCHENKO, A. 2006. Terminator: Beyond Safety. In *Proc. 18th International Conference on Computer Aided Verification (CAV '06)*. 415–418.

COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2005. The astrée analyser. In *Proc. 14th European Symposium On Programming (ESOP '05)*. 21–30.

CRAMER, R. AND SHOUP, V. 2000. Signature Schemes Based on the String RSA Assumption. *ACM Transactions on Information and System Security 3,* 3, 161–185.

DEBRAY, S. K., MUTH, R., AND WEIPPERT, M. 1999. Alias analysis of executable code. In *Proc. 26th Symposium on Principles of Programming Languages (POPL '99)*. 12–24.

DOLEV AND YAO, A. 1981. On the security of public key protocols. In *Proc. of the IEEE 22nd Annual Symposium on Foundations of Computer Science (FOCS)*. 350–357.

DOLEV, D., DWORK, C., AND NAOR, M. 2000. Non-Malleable Cryptography. *SIAM Journal of Computing (SIAMJC) 30,* 2, 391–437.

ELGAMAL, T. 1985. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory 31,* 4, 469–472.

GOLDREICH, O. 2002. Secure multi-party computation. Final Draft, Version 1.4.

GOLDREICH, O. 2004. *Foundations of Cryptography*. Vol. II: Basic Applications. Cambridge University Press.

GOTSMAN, A., BERDINE, J., AND COOK, B. 2006. Interprocedural Shape Analysis with Separated Heap Abstractions. In *Proc. 13th International Static Analysis Symposium (SAS '06)*. 240–260.

HEINECKE, H. 2004. Automotive Open System Architecture-An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures. Tech. Rep. 2004-21-0042, Society of Automotive Engineers.

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy Abstraction. In *Proc. 29th Symposium on Principles of Programming Languages (POPL '02)*. 58–70.

KINDER, J. AND VEITH, H. 2010. Precise static analysis of untrusted driver binaries. In *Proc. 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*. 43–50.

LEE, D. AND YANNAKAKIS, M. 1994. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers 43,* 3, 306–320.

LEE, D. AND YANNAKAKIS, M. 1996. Principles and Methods of Testing Finite State Machines – a Survey. *Proceedings of the IEEE 84,* 8, 1090–1126.

NECULA, G. C. 1997. Proof-Carrying Code. In *Proc. 24th Symposium on Principles of Programming Languages (POPL '97)*. 106–119.

NIST. 1995. NIST FIPS PUB 180-1, Secure Hash Standard.

PELED, D., VARDI, M. Y., AND YANNAKAKIS, M. 1999. Black box checking. In *Proc. 19th Conference on Formal Description Techniques for Networked and Distributed Systems (FORTE '99)*. 225–240.

PETITCOLAS, F. AND KATZENBEISSER, S., Eds. 2000. *Information Hiding Techniques for Steganography and Digital Watermarking*. Artech House.

PODELSKI, A. AND RYBALCHENKO, A. 2007. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *Proc. 9th International Symposium Practical Aspects of Declarative Languages*. Lecture Notes in Computer Science (LNCS) Series, vol. 4354. 245–259.

PRETSCHNER, A., BROY, M., KRÜGER, I., AND STAUNER, T. 2007. Software Engineering for Automotive Systems: A Roadmap. In *Proc. Future of Software Engineering (FOSE '07)*. 55–71.

RAVI, S., RAGHUNATHAN, A., AND CHAKRADHAR, S. T. 2004. Tamper resistance mechanisms for secure, embedded systems. In *Proc. 17th International Conference on VLSI Design*. 605–611.

REPS, T. W., BALAKRISHNAN, G., LIM, J., AND TEITELBAUM, T. 2005. A next-generation platform for analyzing executables. In *Proc. 3rd Asian Symposium on Programming Languages and Systems (APLAS '05)*. 212–229.

RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. M. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM (CACM) 21,* 2, 120–126.

SAGIV, S., REPS, T. W., AND WILHELM, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and System (TOPLAS) 24,* 3, 217–298.

WIEDIJK, F., Ed. 2006. *The Seventeen Provers of the World*. Lecture Notes in Computer Science Series, vol. 3600.