

Verification Across Intellectual Property Boundaries

SAGAR CHAKI

Software Engineering Institute

and

CHRISTIAN SCHALLHART

Oxford University

and

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

Author's address: chaki@sei.cmu.edu, christian.schallhart@comlab.ox.ac.uk, veith@forsyte.at

A preliminary version of this paper appeared at CAV 2007 [Chaki et al. 2007].

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.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2001 ACM 0000-0000/2001/0000-0001 \$5.00

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.

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 pri-

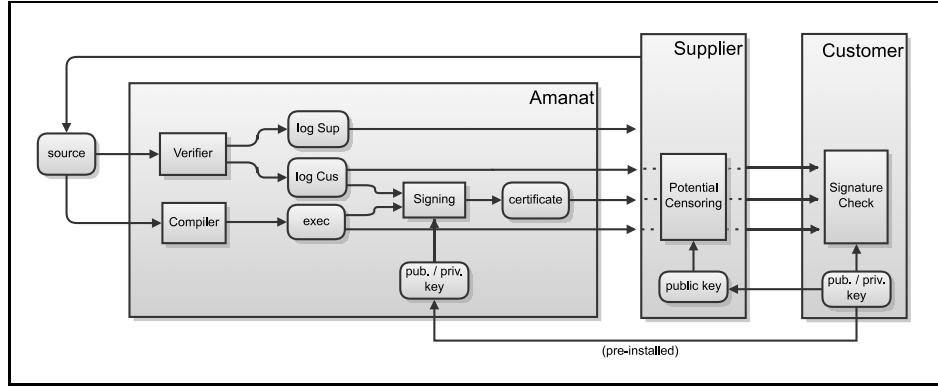


Fig. 1. A High-Level View of the Amanat Protocol

vate 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 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.

1.1 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; Podelski and Rybalchenko 2007]. 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 [Cook et al. 2006] and Slayer [Gotsman et al. 2006] do not require specifications as they are built to verify specific critical properties – termination and memory-safety, respectively.

Software engineering has become essential in the automotive industries: For example, the current BMW 7 series implements about 270 user observable features deployed on 67 embedded platforms with approximately 65 MB of binary code [Pretschner et al. 2007]. In 2002, an estimated value of € 127 billion was created by electric, electronics, and software components within the automotive domain—and by 2015, this amount is expected to rise up to € 316 billion [Dannenberg and Kleinhans 2004]. As evolution and integration have been identified as predominant issues in automotive system software [Pretschner et al. 2007], a number of initiatives have been started to establish standardized and industry-wide accepted computing environments, most notably the Automotive Open System Architecture (AUTOSAR) [07: 2007a], the Open Systems and the Corresponding Interfaces for Automotive Electronics (OSEK)¹ [07: 2007c], and the Japan Automotive Software Platform Architecture (JASPAR) [07: 2007b]. These standardization efforts originate in the need to *integrate software components developed by various companies along a deep supply chain*. This task demands for verification of both, general requirements to be satisfied by every component and specific requirements dedicated to individual components—in presence of heavy IP interests which must be respected and protected [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. For example, the amanat protocol is applicable to the following techniques:

- (1) apply static analysis tools such as ASTREE [Cousot et al. 2005] and TVLA [Sagiv et al. 2002].
- (2) check the correctness of a proofs provided by Sup, given in, e.g., PVS, ISABELLE, Coq or another prover [Wiedijk 2006].
- (3) evaluate worst case execution times experimentally [Wenzel et al. 2005] or statically [Ferdinand et al. 2004].
- (4) generate white box test cases [Holzer et al. 2008; 2010], and execute them.

¹The acronym is derived from the German project name “Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen”.

- (5) validate that the source code comes with a test suite which satisfies previously agreed coverage criteria, using e.g. CoverageMeter [CMeter] or BullseyeCoverage [Bullseye].
- (6) check that the source code is syntactically safe, e.g. using LINT [Darwin 1986].
- (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 code is well-documented.
- (12) validate the development steps by analyzing the CVS or SVN tree.
- (13) ensure compatibility of the source code with language standards.

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—using the tool Verifier alone—that source and the resulting binary exec are indeed satisfying the specified requirements. To this end, source may also contain auxiliary information supporting Verifier in computing the verification verdict. For example, if Verifier uses a theorem prover (as in item 2 above), then source contains additionally a proof which actually shows the conformance of the source code to the specification. For most cases mentioned above, at least some auxiliary information is provided in source, such as command line options, abstraction functions, or test cases.

It is essential that Ama (more precisely, the agreed upon tool Verifier) only accepts auxiliary information which helps in proving the required correctness properties more efficiently but does not take influence on the resulting verdict. For example, a proof for a theorem prover will help Ama to verify the required properties—but if the provided proof is wrong or misleading then Ama will not produce a wrongly positive verdict.

1.2 Security of the Amanat Protocol

We present in Sections 4 and 5 cryptographic proofs for the *secrecy* and *conformance* of the amanat verification protocol. Stronger than term-based proofs in the Dolev-Yao model [Dolev and Yao 1981], 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 (for a technical definition of negligible probability, see the discussion following Definition 7). The practical security of the protocol is also ensured by its simplicity: As the protocol is based on well-known cryptographic encryption and signing schemes, it can be implemented with reasonable effort.

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, the cryptographic proofs need to assume a physical system configuration and infrastructure 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 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, which acts as 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 and requires no expertise beyond server hosting. For the supplier, scenario A has the disadvantage that the encrypted source 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, e.g., regular checks by the customer, a third party, a traditional alarm system, or the use of sealed hardware [Ravi et al. 2004]. All communication channels of Ama are hardwired to the communication filter of the supplier.

We believe that in our B2B settings, scenario B is practically feasible: It only requires that the seal is checked after verification and before deployment. 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 send messages at specific time points and to thereby leak information. *The supplier can read, delay, drop, and modify all outgoing messages*—which is a convincing and non-technical 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 shown in Sections 4 and 5, respectively, and the paper is concluded with Section 6.

While the proof on secrecy follows an intuitive argument, the proof of conformance is technically more involved. We therefore start Section 5 with the theorem stating the conformance of the protocol, even before precisely defining the underlying cryptographic assumptions. Subsequently, we introduce these assumptions in Section 5.1, give an overview on the proof in Section 5.2 and present its details in Sections 5.3 to 5.6.

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]), 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, and malicious code detection [Christodorescu et al. 2005; Kinder et al. 2008]. 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 [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. 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. 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 aims to resolve 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

To make the protocol requirements more precise, we fix some notation and assumptions about the tool landscape. We restrict our tools to run in deterministic polynomial time to ensure that their computations can be efficiently reproduced. When we 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 1 (Compiler) *The compiler Compiler translates an input source into an executable $\text{exec} = \text{Compiler}(\text{source})$ 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 2 (Verification Tool) *The verification tool Verifier takes the input source and computes in deterministic polynomial time two verification verdicts, \log_{Sup} and \log_{Cus} , i.e., $\langle \log_{\text{Sup}}, \log_{\text{Cus}} \rangle = \text{Verifier}(\text{source})$.*

Here, \log_{Sup} is a detailed verdict for the supplier possibly containing IP-critical information such as counterexamples or witnesses for certain properties. The second output \log_{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. Thus, the specification is included into source—allowing Ama to output the verification results together with their specifications into \log_{Cus} . Hence, Sup can check which properties have been verified by Ama. All auxiliary information necessary to run Verifier is provided by Sup as part of source, such as command line parameters or abstraction functions.

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 $(\log_{\text{Sup}}, \log_{\text{Cus}})$ of combined outputs.

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

Definition 3 (Conformance) *An execution of the amanat protocol is conformant, if the delivered binary exec and verdict \log_{Cus} have been produced from the same source.*

Definition 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_{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.

We note that some of the possible verification tasks discussed in Section 1—in particular 7, 10, 11—are concerned with non-functional properties of the source code which do not affect the executable produced by the compiler. However, the conformance property *only* proves to the customer that the obtained binary and its verification verdict stem from the same source code. Thus, in the case of a legal conflict, a court can *only* require the supplier to provide a source code which (i) compiles into the purchased executable, and (ii) produces the same verification output \log_{Cus} . There is no mathematical guarantee however, that the revealed code will be *identical* to the original code. For example, if the verdict only indicates whether the delivered executable is deadlock free or not, then the supplier can decide to not reveal its original source but to provide an obfuscated source which does not contain any comments and has all identifiers renamed into random strings.

Remark 1 (Preventing Obfuscation) *In situations where it is necessary to obtain the original source, we equip Verifier with (i) a check that source does not have an obfuscated appearance, and (ii) a secure hash computation which appends the hash of source to \log_{Cus} —leaving the protocol itself unchanged. Then, in case of a conflict, Sup must provide its original source to match the secure hash.*

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 $\langle k_{priv}, k_{pub} \rangle$ can be used for both. This can be easily achieved by combining the key-pairs for both schemes into a single pair.

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

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

Aside providing these interfaces, suitable cryptographic primitives must also satisfy the relevant security properties, as defined in Section 5.1: 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 \log_{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: Cus initializes Ama with a master key pair $\langle km_{priv}, km_{pub} \rangle$ which is used later to exchange a session key pair. Then, Ama is transported to and installed at the designated site. All further 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: After installation, Sup and Cus must agree on a specific Verifier and Compiler. Once Verifier and Compiler have been fixed, the session initializa-

tion phase starts: First, Cus generates a new pair of session keys $\langle ks_{priv}, ks_{pub} \rangle$ and sends them to Ama via Sup—having encrypted the private key ks_{priv} beforehand. Then, the new session keys are used to produce certificates $cert_{Verifier}$ and $cert_{Compiler}$ for Verifier and Compiler, respectively. Sup checks the contents of the certificates. If they are valid, it uses them to setup Ama with Verifier and Compiler. Ama in turn accepts Verifier and Compiler if their certificates are valid.

In the last initialization step, Ama generates a supply of random seeds R_1, \dots, R_t for t subsequent executions of the certification phase. It also generates a sequence of key pairs $\langle kr_{priv}^1, kr_{pub}^1 \rangle, \dots, \langle kr_{priv}^t, kr_{pub}^t \rangle$ for each random seed R_i . Ama finally encrypts each random seed to obtain and send $kr_{pub}^i(R_i)$ 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.

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
 — $cert_{Verifier} = csign(ks_{priv}, Verifier)$ and
 — $cert_{Compiler} = csign(ks_{priv}, Compiler)$.
 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
 — $cverify(ks_{pub}, Verifier, cert_{Verifier})$ and
 — $cverify(ks_{pub}, Compiler, cert_{Compiler})$ succeed.
 If one of the checks fails, Sup aborts the protocol.

S4 Amanat Tool Transmission [Sup]

Sup sends Verifier, Compiler, and the certificates $cert_{Verifier}$ and $cert_{Compiler}$ to Ama.

S5 Amanat Validation of the Tool Certificates [Ama]

Ama checks whether Verifier and Compiler are properly certified, i.e., it checks whether
 — $cverify(ks_{pub}, Verifier, cert_{Verifier})$ and
 — $cverify(ks_{pub}, Compiler, cert_{Compiler})$ 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, \dots, R_t together with a series of corresponding key pairs $\langle kr_{priv}^1, kr_{pub}^1 \rangle, \dots, \langle kr_{priv}^t, kr_{pub}^t \rangle$,
 —encrypts the random seeds with the corresponding keys $kr_{pub}^i(R_i)$ for $i = 1, \dots, t$, and
 —initializes round counter $round = 0$.
 Ama sends $kr_{pub}^i(R_i)$ and kr_{pub}^i for $i = 1, \dots, t$ 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.

During certification, Ama runs Verifier and Compiler on source and generates a certificate cert for the binary exec and the verification output \log_{Cus} dedicated to Cus. The certificate is based upon the random seed R_{round} which Ama committed to use in this round during session initialization. Ama sends the certificate cert, the outputs \log_{Sup} and \log_{Cus} , and the key $kr_{\text{priv}}^{\text{round}}$ to Sup.

To validate secrecy, Sup computes the random seed $R_{\text{round}} = kr_{\text{priv}}^{\text{round}}(kr_{\text{pub}}(R_{\text{round}}))$ which Ama supposedly used for the generation of cert. Then Sup checks that the certificate cert is valid and based upon the random seed R_{round} . If this is the case, Ama cannot hide any unintended information in the certificates. Otherwise, if the checks fails, Sup aborts the protocol. If Sup proceeds and the obtained verdict is good enough, it forwards the results to Cus. Otherwise, it might revise source and start a new certification phase. Finally, when Cus receives exec, \log_{Cus} , and cert, it checks conformance exec and \log_{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 $\langle \log_{\text{Sup}}, \log_{\text{Cus}} \rangle = \text{Verifier}(\text{source})$ of Verifier on source,
—the binary exec = Compiler(source),
—increments the round counter round, and
—cert = $\text{csign}(ks_{\text{priv}}, \langle \text{exec}, \log_{\text{Cus}} \rangle, R_{\text{round}})$.
Ama sends exec, \log_{Sup} , \log_{Cus} , cert, and $kr_{\text{priv}}^{\text{round}}$ to Sup.

C3 Secrecy Validation [Sup]
Upon receiving exec, \log_{Sup} , \log_{Cus} , cert, and $kr_{\text{priv}}^{\text{round}}$, Sup
—decrypts the random seed with $R_{\text{round}} = kr_{\text{priv}}^{\text{round}}(kr_{\text{pub}}^{\text{round}}(R_{\text{round}}))$,
—checks whether exec = Compiler(source) and $\langle \log_{\text{Sup}}, \log_{\text{Cus}} \rangle = \text{Verifier}(\text{source})$ hold, and
—verifies that $\text{cverify}(ks_{\text{pub}}, \langle \text{exec}, \log_{\text{Cus}} \rangle, \text{cert}, R_{\text{round}})$ 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_{Cus} and \log_{Sup} and decides whether to deliver exec, \log_{Cus} , and cert to Cus in step C4 or whether to abort the protocol.

C4 Conformance Validation [Cus]
Having received exec, \log_{Cus} , and cert, Cus verifies that certificate is valid, i.e., that $\text{cverify}(ks_{\text{pub}}, \langle \text{exec}, \log_{\text{Cus}} \rangle, \text{cert})$ 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_{Cus} and decides whether the verification verdict supports the purchase of the product exec.

4. PROTOCOL SECRECY

We designed the amanat protocol aiming at a simple and intuitive argument for its secrecy, as such a straightforward proof is a prerequisite to convince a code supplying company

that the protocol keeps their highly valued IP assets safe. For the same reason, we do not rely on any cryptographic assumptions to prove the secrecy of the amanat protocol.

The idea behind this proof is straightforward: 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 1 (Secrecy) *The amanat protocol enforces secrecy (see Definition 4) in all its executions unconditionally.*

This means that Cus is unable to extract any piece of information on the source source which is not contained in exec and \log_{Cus} in any case and independently from any cryptographic assumptions.

PROOF. During the execution of the protocol, Cus receives the binary exec , the output file \log_{Cus} , and the certificate cert . The certificate $\text{cert} = \text{csign}(ks_{\text{priv}}, \langle \text{exec}, \log_{\text{Cus}} \rangle, R_{\text{round}})$ is generated from exec , \log_{Cus} , the key ks_{priv} , and the random seed R_{round} . Cus generates ks_{priv} itself and obtains access to exec and to \log_{Cus} . Thus the only additional information communicated from Ama to Sup stems from the random seed R_{round} . But R_{round} has been fixed by Ama before having access to source and only depends on the iteration counter round . Thus, Ama cannot encode any information from source into R_{round} —such that the certificate depends on exec and \log_{Cus} only. \square

Note that the order of the random seeds R_1, \dots, R_t must be predetermined: Otherwise, Ama could choose a random seed R_i *after* evaluating source according to some conspirative scheme Ama and Cus agreed upon.

5. PROTOCOL CONFORMANCE

We prove 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]. Assuming these security properties, we obtain the conformance of our protocol under all practically relevant conditions.

Theorem 2 (Conformance) *If the protocol terminates (in Step C4 of the certification phase) with the customer Cus accepting the certificate, then the protocol execution has been conformant in all but a negligible fraction of the cases.*

We assume that (a) the underlying encryption is semantically secure, (b) the signature scheme is secure against adaptively chosen message attacks, and (c) the supplier Sup runs in probabilistic polynomial time.

To prove this theorem, we recall in Section 5.1 the stated security properties, discuss in Section 5.2 the proof structure, and concretize the proof throughout Sections 5.3 to 5.6.

5.1 Security Properties

Semantic security means that a probabilistic polynomial time adversary cannot learn more from the ciphertext than from the length of the plaintext alone.

Definition 7 (Semantic Security [Goldreich 2004]) A public-key encryption scheme (Definition 5) is semantically secure if for every probabilistic polynomial time algorithm A , there exist a probabilistic polynomial time algorithm A' and an integer N , such that for every choice of X_n , f , h , p , $n \geq N$, and randomly chosen public key k_{pub} ,

$$\Pr [A(1^n, k_{pub}, k_{pub}(X_n), 1^{|X_n|}, h(1^n, X_n)) = f(1^n, X_n)] \\ < \Pr [A'(1^n, 1^{|X_n|}, h(1^n, X_n)) = f(1^n, X_n)] + 1/p(n)$$

holds. Therein, X_n is a random variable of arbitrarily distributed plaintexts, f and h are functions in the security parameter n and the plaintext X_n which both yield a result of polynomial length, and p is a polynomial.

In this definition, we have two procedures A and A' , where A' receives the same information as A —except for the public key k_{pub} and the ciphertext $k_{pub}(X_n)$ which are dropped.

In many attacks on a public-key encryption scheme, the attacker does not only receive some ciphertexts but also obtains some related information. To model this situation, the function $h(1^n, X_n)$ is used to provide both, A and A' , with further information depending on the plaintext X_n .

Then an encryption scheme is defined as semantically secure, if for every A there exists an A' such that the probability that A and A' differ in their results is at most negligible, i.e., both compute the same function up to a negligible fraction of the cases. Thereby, we say that a function $f : \mathbb{N} \rightarrow \mathbb{R}$ is *negligible*, iff there exists an integer N for every polynomial p such that for all $n \geq N$, $f(n) < 1/p(n)$ holds. When a probabilistic experiment parameterized with n has *negligible success probability*, then we mean that this success probability as function in the parameter n is negligible. See [Goldreich 2004] for more details.

An *adaptive chosen message attack* is an attack against a signature scheme, where the attacker is given a public key and access to an oracle which can sign arbitrary messages with the corresponding private key. The attacker generates a number of messages to be signed by the oracle. The generated messages may depend on each other, on the public key itself, and—since the attack is adaptive—on the signatures previously returned by the oracle. The attack procedure must then compute a message and a corresponding signature which has *not been signed before by the oracle*.

If every probabilistic polynomial time attacker is only successful in forging a signature with a *negligible probability*, then we say that the signature scheme is *secure against adaptive chosen message attacks*.

Definition 8 (Adaptive Chosen Message Attack [Cramer and Shoup 2000]) Given a public key-pair $\langle k_{priv}, k_{pub} \rangle$ for a signature scheme, a signing oracle $S[k_{priv}]$ with private key k_{priv} is a function which takes a message m and returns a signature $s = \text{csign}(k_{priv}, m, R)$ for a uniformly and randomly chosen random seed R . A forging algorithm $F(k_{pub})$ receives the public key k_{pub} and has access to the signing oracle $S[k_{priv}]$, where k_{priv} is the private key corresponding to k_{pub} .

The algorithm F is allowed to query $S[k_{priv}]$ 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 $\text{cverify}(k_{pub}, m, s)$ succeeds,

although m has never been sent to and signed by $S[k_{priv}]$.

A signature scheme (Definition 6) is secure against adaptive chosen message attacks, if every probabilistic polynomial time algorithm F has only a negligible success probability.

Assuming that the encryption scheme is semantically secure and that the signature scheme is secure against adaptive chosen message attacks, we start the conformance proof.

5.2 Proof Overview

For the sake of contradiction, we assume that Sup is able to trick Cus into accepting a forged pair $\langle \text{exec}, \log_{Cus} \rangle$ with a not negligible success probability, i.e., Sup computes a certificate cert for a pair $\langle \text{exec}, \log_{Cus} \rangle$ which has not been computed and signed by Ama but is nevertheless accepted by Cus. Starting from this assumption, we derive a contradiction in four steps.

1. *Modeling the Supplier (Section 5.3).* As a first step in the proof, we introduce in Fact 1 and 2 two procedures Sup_gen and Sup_val which model the computations of a honest supplier in Steps C1 and C3 respectively. Then each step of the supplier Sup corresponds to a single call to one of these two procedures: In each call, we provide the procedure with all information which has been made available to Sup during the protocol execution so far. After each call, the procedures Sup_gen and Sup_val return all information which is necessary to continue the protocol execution. Since Sup can maintain an internal state between the individual protocol steps, we also introduce a variable state which is given to both procedures Sup_gen and Sup_val by reference, i.e., they can access and update this variable state.

2. *A first Forging Procedure (Section 5.4).* Since a malicious supplier is assumed to exist, there must exist a corresponding pair of procedures MSUp_gen and MSUp_val implementing the malicious behavior of this supplier: These procedures MSUp_gen and MSUp_val are only restricted to support the same interface as Sup_gen and Sup_val and to respect a probabilistic polynomial time bound. We use MSUp_gen and MSUp_val to build a first forging procedure Forge₁: This procedure simulates the possibly repeated certificate phase of the Amanat protocol such that MSUp_gen and MSUp_val cannot distinguish this simulation from a real protocol execution. Since they are unable to distinguish the simulation from a real protocol execution, they will behave exactly the same way—and consequently, in Proposition 1 we conclude that Forge₁ produces the same result as a real protocol execution. In the remainder of the proof, this procedure is transformed via an intermediate procedure Forge₂ into the final attack algorithm Forge₃.

3. *Removing the Random Seeds and the Private Key (Section 5.5).* The procedure Forge₁ is not directly usable to forge certificates, since it uses the private ks_{priv} which is supposed to be unknown to Forge₁. Hence, we want to use the signing oracle $S[ks_{priv}]$ instead; but when we rely on $S[ks_{priv}]$ to generate certificates, we must also avoid referring to the yet unused certificate random seeds $R_{round+1}, \dots, R_t$ (since the choices of $S[ks_{priv}]$ for these seeds are unpredictable). Within Forge₁, MSUp_gen and MSUp_val both receive the session key and the random seeds in an encrypted manner, i.e., they take $km_{pub}(ks_{priv})$ and $kr_{pub}^{round+1}(R_{round+1}), \dots, kr_{pub}^t(R_t)$ as arguments. Because ks_{priv} and $R_{round+1}, \dots, R_t$ are passed on in an encrypted manner, we can apply semantic security to remove the respective arguments (Lemmata 1 and 2). In particular, we prove the existence of two

procedures $\overline{\text{MSup_gen}}$ and $\overline{\text{MSup_val}}$: They receive the same arguments as their original counterparts—except for the encrypted session key and the encrypted, yet unused random seeds—but return the same output as MSup_gen and MSup_val , respectively. Next, in Lemma 3 and Corollary 1, we show that we can substitute $\overline{\text{MSup_gen}}$ and $\overline{\text{MSup_val}}$ for MSup_gen and MSup_val in any probabilistic polynomial time procedure—again without changing the result of the procedure. Lemmata 1 to 3 hold in all but a negligible fraction of the cases.

4. *The Proof of Theorem 2 (Section 5.6).* Lemma 3 applies to Forge_1 since it is a probabilistic polynomial time procedure. Therefore, we replace in Forge_1 MSup_gen and MSup_val with their counterparts $\overline{\text{MSup_gen}}$ and $\overline{\text{MSup_val}}$ to obtain Forge_2 . Since Forge_2 uses the private key ks_{priv} for signing only and does not make any use of the random seeds $R_{\text{round}+1}, \dots, R_t$, we can replace all references to ks_{priv} with its signing oracle $S[ks_{priv}]$ to obtain Forge_3 . This procedure Forge_3 is the sought for adaptive chosen message attack—contradicting the assumption that no such attack exists.

In the proof below, we assume that *all procedures receive the security parameter 1^n implicitly as their first parameter*. We also assume that their computations are polynomially bounded in n and the length of their other inputs.

5.3 Modeling the Supplier

To introduce the two procedures Sup_gen and Sup_val which perform all computations of the supplier Sup during the certification phase, we observe that the supplier receives the following pieces of information during the installation, session initialization, and *possibly repeated* certification phase:

- The encrypted random seeds $kr_{pub}^i(R_i)$ and the respective public keys kr_{pub}^i for $i = 1, \dots, t$ are sent to Sup in Step **S6**.
- The accumulated inputs source , exec , \log_{Sup} , \log_{Cus} , cert , and kr_{priv}^{round} are sent to Sup at the end of Step **C2** of all previous certification phases.
- All remaining messages of the initialization and session initialization phase comprise the following information: km_{pub} (**I2**), $km_{pub}(ks_{priv})$ and ks_{pub} (**S1**), as well as $\text{cert}_{\text{Verifier}}$ and $\text{cert}_{\text{Compiler}}$ (**S2**).

We model the accumulated state of Sup with an additional variable state. This variable state is given to Sup_gen and Sup_val by *reference*, i.e., both procedures can access and update its contents. Initially, state contains all messages seen or generated by Sup during the initialization and session initialization phase (km_{pub} , ks_{pub} , $\text{cert}_{\text{Compiler}}$, and $\text{cert}_{\text{Verifier}}$). The encrypted key $km_{pub}(ks_{priv})$ is handled explicitly and is therefore not added to state. During a number of repeated certification phases, Sup can also accumulate and store information on the received instances of source , exec , \log_{Sup} , \log_{Cus} , and cert in state.

Fact 1 (Modeling Sup for Step **C1: Sup_gen)** *The computation of the supplier Sup in Step **C1** of the certification phase are modeled with a call*

$$\text{source} = \text{Sup_gen} \left(\text{state}, km_{pub}(ks_{priv}), \right. \\ \left. kr_{pub}^1, \dots, kr_{pub}^t, \right. \\ \left. kr_{pub}^1(R_1), \dots, kr_{pub}^t(R_t), \right. \\ \left. kr_{priv}^1, \dots, kr_{priv}^{\text{round}} \right)$$

where state is the state of Sup , $km_{pub}(ks_{priv})$ is the encrypted session key, round is the protocol parameter being incremented with each iteration of the certification phase, and t is the total number of precomputed random seeds.

Fact 2 (Modeling Sup for Step C3: Sup_val) *The computation of the supplier Sup in Step C3 of the certification phase can be modeled with a call*

$$\text{result} = \text{Sup_val}(\text{state}, km_{pub}(ks_{priv}), \text{cert}, \\ kr_{pub}^1, \dots, kr_{pub}^t, \\ kr_{pub}^1(R_1), \dots, kr_{pub}^t(R_t), \\ kr_{priv}^1, \dots, kr_{priv}^{\text{round}})$$

where state is the state of Sup , $km_{pub}(ks_{priv})$ is the encrypted session key, cert is the certificate produced for the formerly generated source, round is the protocol parameter being incremented with each iteration of the certification phase, t is the total number of precomputed random seeds, and result indicates either

- to continue the protocol at Step C4 (in this case result contains a pair $\langle \text{exec}, \log_{\text{Cus}} \rangle$ and a corresponding but possibly forged cert to be sent to the Cus),
- to abort the protocol and start again at Step C1, or
- to refuse to work with Ama since a secrecy violation has been detected.

In Fact 2, Sup_val is only receiving cert while Sup receives in Step C2 exec , \log_{Sup} , and \log_{Cus} , as well. However, Sup_val can compute these pieces of information itself with $\text{exec} = \text{Compiler}(\text{source})$ and $\langle \log_{\text{Sup}}, \log_{\text{Cus}} \rangle = \text{Verifier}(\text{source})$. To do so, Sup_gen stores source in state .

Note that we have to call Sup_gen and Sup_val in an alternating manner in order to ensure that these two procedures cannot distinguish a simulation from a real protocol execution: This is necessary since state can be used to communicate information from Sup_gen to Sup_val and since they are called in an alternating manner in the original protocol.

5.4 A first Forging Procedure

We assume that there exists a malicious supplier MSup which produces a pair $\langle \text{exec}, \log_{\text{Cus}} \rangle$ and a certificate cert within probabilistic polynomial time at a *not negligible probability* such that

- Cus accepts the pair and its certificate in Step C4, but
- Ama did not produce the certificate cert for $\langle \text{exec}, \log_{\text{Cus}} \rangle$.

We model this malicious supplier with two procedures MSup_gen and MSup_val , which run in probabilistic polynomial time and have the same interface as Sup_gen and Sup_val , as described in Facts 1 and 2.

Now we use the two malicious procedures to build a probabilistic polynomial time forging algorithm Forge_1 . More specifically, Forge_1 embeds MSup_gen and MSup_val in a simulated execution of the amanat protocol, consisting of a single execution of the session initialization phase and a possibly repeated execution of the certification phase. Since MSup_gen and MSup_val cannot distinguish the simulation from the real protocol execution, the success probability of Forge_1 is identical to the success probability of the original malicious supplier. More specifically, the procedure Forge_1 takes as input a session

key-pair $\langle ks_{priv}, ks_{pub} \rangle$ and—given the malicious character of MSUp_gen and MSUp_val—produces with not negligible probability a forged signature for this key-pair: After computing with ks_{priv} a number of signatures, Forge₁ uses the accumulated information to *forge* a signature, i.e., it computes a signature without using ks_{priv} . However, MSUp_gen and MSUp_val do require access to the encrypted session key and random seeds—which is the problem addressed in the following sections.

F1₁ Key and Random Seed Generation

Simulate the session initialization phase:

- Generate a sequence $\langle kr_{priv}^1, kr_{pub}^1 \rangle, \dots, \langle kr_{priv}^t, kr_{pub}^t \rangle$ of key pairs.
- Generate a sequence R_1, \dots, R_t of random seeds.
- Generate a master key pair $\langle km_{priv}, km_{pub} \rangle$.
- Set round = 0.
- Initialize state with km_{pub} , ks_{pub} , cert_{Compiler}, and cert_{Verifier}.

F2₁ Source Code Computation

Simulate Step C1 with a call

$$\text{source} = \text{MSUp_gen} \left(\text{state}, km_{pub}(ks_{priv}), \right. \\ \left. kr_{pub}^1, \dots, kr_{pub}^t, \right. \\ \left. kr_{pub}^1(R_1), \dots, kr_{pub}^t(R_t), \right. \\ \left. kr_{priv}^1, \dots, kr_{priv}^{\text{round}} \right).$$

F3₁ Source Code Verification

Simulate Step C2 by computing

- $\langle \log_{\text{Sup}}, \log_{\text{Cus}} \rangle = \text{Verifier}(\text{source})$ and
 - $\text{exec} = \text{Compiler}(\text{source})$,
 - incrementing round by 1, and
 - computing the certificate
- $$\text{cert} = \text{csign}(ks_{priv}, \langle \log_{\text{Cus}}, \text{exec} \rangle, R_{\text{round}}).$$

F4₁ Secrecy Validation (Forge Certificate)

Simulate Step C3, i.e., make a call

$$\text{result} = \text{MSUp_val} \left(\text{state}, km_{pub}(ks_{priv}), \text{cert}, \right. \\ \left. kr_{pub}^1, \dots, kr_{pub}^t, \right. \\ \left. kr_{pub}^1(R_1), \dots, kr_{pub}^t(R_t), \right. \\ \left. kr_{priv}^1, \dots, kr_{priv}^{\text{round}} \right).$$

Depending on result, the execution proceeds with

- Step F5₁, if result indicates to continue the protocol (in this case result contains $\langle \text{exec}, \log_{\text{Cus}} \rangle$ and cert, where cert is forged with not negligible probability).
- Step F2₁, if result indicates to start over again.
- an **erroneous abort**, if result indicates that a secrecy violation has been detected.

F5₁ Output Result

Output the pair $\langle \text{exec}, \log_{\text{Cus}} \rangle$ and cert as indicated by result.

The procedure Forge₁ simulates the repeated execution of the certification phase after a preceding initialization and session initialization phase. The procedure MSUp_gen and MSUp_val cannot distinguish between the protocol execution and the simulation within

Forge₁ and therefore, Forge₁ produces a forged certificate in Step **F5₁** with the same probability as the original supplier MSup in a protocol execution. This leads to the following proposition:

Proposition 1 (The Procedure Forge₁) *Let MSup_{gen} and MSup_{val} model a malicious supplier which is able to produce a non-conformant pair $\langle \text{exec}, \log_{\text{Cus}} \rangle$ with a valid (i.e., forged) certificate cert with not negligible probability. Then, with non negligible probability, the procedure Forge₁ outputs a forged certificate in Step **F5₁**, i.e., one that has not been signed before in Step **F3₁**.*

However, both procedures MSup_{gen} and MSup_{val} are provided with the encrypted session key $km_{pub}(ks_{priv})$ and the sequence $kr_{pub}^1(R_1), \dots, kr_{pub}^t(R_t)$, i.e., Sup receives in advance all random seeds which are used subsequently for certificates. Since we want to use this procedure to mount an adaptive chosen messages attack (Definition 8), we need to replace all direct references to ks_{priv} with queries to a signing oracle $S[ks_{priv}]$. But a signing oracle chooses its random seeds independently, and therefore we have to cut down the sequence $kr_{pub}^1(R_1), \dots, kr_{pub}^t(R_t)$ to $kr_{pub}^1(R_1), \dots, kr_{pub}^{\text{round}}(R_{\text{round}})$, i.e., no information on the random seeds $R_{\text{round}+1}, \dots, R_t$ is allowed to be received by Sup in advance.

5.5 Removing the Random Seeds and the Private Key

As the random seeds $kr_{pub}^{\text{round}+1}(R_{\text{round}+1}), \dots, kr_{pub}^t(R_t)$ and the private key $km_{pub}(ks_{priv})$ are all encrypted, we can use the assumption that the encryption scheme is semantically secure, as defined in Definition 7. We first deal with MSup_{gen} and then draw analogous conclusions for MSup_{val}.

Lemma 1 (Removing Encrypted Seeds: $\overline{\text{MSup}_{\text{gen}}}$) *For each probabilistic polynomial time procedure MSup_{gen} which implements the signature of Fact 1, there exists another probabilistic polynomial time procedure $\overline{\text{MSup}_{\text{gen}}}$ with the signature*

$$\text{source} = \overline{\text{MSup}_{\text{gen}}}(\text{state}, \\ kr_{pub}^1, \dots, kr_{pub}^{\text{round}}, \\ kr_{pub}^1(R_1), \dots, kr_{pub}^{\text{round}}(R_{\text{round}}), \\ kr_{priv}^1, \dots, kr_{priv}^{\text{round}})$$

such that for each polynomial q and sufficiently large n

$$\Pr [\overline{\text{MSup}_{\text{gen}}}(\dots) = \text{MSup}_{\text{gen}}(\dots)] > 1 - 1/q(n)$$

holds.

PROOF. Fix a procedure MSup_{gen} and a protocol parameter t which is set to the maximum number of certification phases, i.e., the number of pre-committed random seeds. Then we start with the t -th random seed R_t and remove the references to kr_{pub}^t and $kr_{pub}^t(R_t)$. Next, we remove references to kr_{pub}^{t-1} and $kr_{pub}^{t-1}(R_{t-1})$, and so forth, until we reach kr_{pub}^{round} and $kr_{pub}^{\text{round}}(R_{\text{round}})$.

To apply Definition 7, we define h as constant function with

$$h(R_t) = (\text{state}, km_{pub}(ks_{priv}), \\ kr_{pub}^1, \dots, kr_{pub}^{t-1}, \\ kr_{pub}^1(R_1), \dots, kr_{pub}^{t-1}(R_{t-1}), \\ kr_{priv}^1, \dots, kr_{priv}^{t-1}).$$

After permuting some arguments of `MSup_gen`, we can rewrite a call to `MSup_gen` as

$$\text{MSup_gen}(kr_{pub}^t, kr_{pub}^t(R_t), h(R_t))$$

and set

$$f(R_t) = \text{MSup_gen}(kr_{pub}^t, kr_{pub}^t(R_t), h(R_t)).$$

Please recall that we omit the implicit security parameter in our procedure headers, since then—by adding the implicit security parameter 1^n again—this setting matches precisely the prerequisites of Definition 7. Thus, by the semantic security of the encryption scheme, there exists a corresponding procedure `MSup_gen'`($h(R_t)$) with

$$\begin{aligned} & \Pr [\text{MSup_gen}(kr_{pub}^t, kr_{pub}^t(R_t), h(R_t)) = f(R_t)] \\ & < \Pr [\text{MSup_gen}'(h(R_t)) = f(R_t)] + \frac{1}{p(n)} \end{aligned}$$

for every polynomial p and sufficiently large n . Since f computes `MSup_gen`, the first probability equals 1, yielding

$$\Pr[\text{MSup_gen}'(h(R_t)) = \text{MSup_gen}(\dots)] > 1 - \frac{1}{p(n)}.$$

Then, for an arbitrary polynomial q , we set $p(n) = (t - \text{round} + 1)q(n)$ and iterate the above process $(t - \text{round})$ times, to obtain—for arbitrary q and sufficiently large n —

$$\begin{aligned} & \Pr[\text{MSup_gen}''(\dots) = \text{MSup_gen}(\dots)] \\ & > \left(1 - \frac{1}{p(n)}\right)^{t-\text{round}} = \left(1 - \frac{1}{(t-\text{round}+1)q(n)}\right)^{t-\text{round}} \end{aligned}$$

where we call `MSup_gen''`(\dots) with signature

$$\text{MSup_gen}'' (\text{state}, km_{pub}(ks_{priv}), \\ kr_{pub}^1, \dots, kr_{pub}^{\text{round}}, \\ kr_{pub}^1(R_1), \dots, kr_{pub}^{\text{round}}(R_{\text{round}}), \\ kr_{priv}^1, \dots, kr_{priv}^{\text{round}}).$$

It remains to remove $km_{pub}(ks_{priv})$ which can be done with one further application of the encryption scheme's semantic security. Then we arrive at

$$\begin{aligned} & \Pr[\overline{\text{MSup_gen}}(\dots) = \text{MSup_gen}(\dots)] \\ & > \left(1 - \frac{1}{(t-\text{round}+1)q(n)}\right)^{t-\text{round}+1} \\ & \geq 1 - \frac{1}{q(n)} \end{aligned}$$

which is the Lemma statement. \square

By substituting MSup_val for MSup_gen , we obtain in a completely analogous manner the next lemma:

Lemma 2 (Removing Encrypted Seeds: $\overline{\text{MSup_val}}$) *For each procedure MSup_val which runs in probabilistic polynomial time and implements the signature of Fact 2, there exists another probabilistic polynomial time procedure $\overline{\text{MSup_val}}$ with the signature*

$$\text{result} = \overline{\text{MSup_val}}(\text{state}, \text{cert}, \\ kr_{pub}^1, \dots, kr_{pub}^{\text{round}}, \\ kr_{pub}^1(R_1), \dots, kr_{pub}^{\text{round}}(R_{\text{round}}), \\ kr_{priv}^1, \dots, kr_{priv}^{\text{round}})$$

such that for each polynomial q and sufficiently large n

$$\Pr [\overline{\text{MSup_val}}(\dots) = \text{MSup_val}(\dots)] > 1 - 1/q(n)$$

holds.

Thus, starting from a malicious supplier which is modeled by two probabilistic polynomial time procedures MSup_gen and MSup_val , we obtain by application of Lemmata 1 and 2 two other probabilistic polynomial time procedures $\overline{\text{MSup_gen}}$ and $\overline{\text{MSup_val}}$ which both

- do not receive any encrypted random seed—besides those which have been used already,
- do not receive the encrypted key $km_{pub}(ks_{priv})$, and
- return the same result as their original counterparts in all but a negligible fraction of the cases.

The same holds true for general polynomial time computations which involve MSup_gen and MSup_val :

Lemma 3 (Substituting $\overline{\text{MSup_gen}}$ and $\overline{\text{MSup_val}}$) *Let A_1 be a probabilistic polynomial time algorithm which invokes MSup_gen and MSup_val (as defined in Fact 1 and 2). Furthermore, let A_2 be the procedure obtained from A_1 by substituting $\overline{\text{MSup_gen}}$ and $\overline{\text{MSup_val}}$ (taken from Lemmata 1 and 2) for MSup_gen and MSup_val , respectively. Then A_2 and A_1 compute deviating results with a negligible probability only.*

PROOF. Since A_1 and A_2 run within polynomial time, they can only invoke their respective subprocedures a polynomial number of times and therefore, their results deviate only if at least one of their polynomial many subprocedure invocations deviates.

But if we repeat an experiment with negligible success probability a polynomial number of times, then the probability of observing at least one successful attempt is still negligible (see for example the introduction of [Goldreich 2004]).

Thus, invoking $\overline{\text{MSup_gen}}$ and $\overline{\text{MSup_val}}$ a polynomial number of times will produce deviations only with negligible probability—and consequently, A_1 and A_2 produce the same result in all but a negligible fraction of the cases. \square

In the proof of Theorem 2, we use the following consequence of Lemma 3:

Corollary 1 (Preserving a Not Negligible Success Prob.) *Let the procedures A_1 and A_2 be given as described in Lemma 3. Then A_1 computes a result successfully with a not negligible probability iff A_2 has a not negligible success probability for the same computation.*

PROOF. We show equivalently that A_1 computes some result with a negligible probability iff A_2 has a negligible success probability for the same computation.

For A_2 to be successful, one of two cases must arise: Either A_1 is successful (and the results of A_1 and A_2 coincide), or the results of A_1 and A_2 deviate (and A_1 is not successful). Thus, the success probability of A_2 is bounded by the probability that either A_1 is successful or that the results of A_1 and A_2 deviate.

The success probability of A_1 is negligible, as well as the probability that the results of A_1 and A_2 deviate. Consequently, the probability that A_2 succeeds is negligible. By exchanging A_1 and A_2 , the converse follows and the statement is proved for both directions. \square

5.6 Proof of Theorem 2

In the final proof, we use Forge_1 , as provided in Section 5.4 and apply Lemma 3 to Forge_1 to obtain the procedure Forge_2 . Then, Forge_2 has the following properties:

- The random seeds to be used in future rounds are not referenced in advance.
- The result computed by Forge_2 deviates from Forge_1 only in a negligible fraction of the cases. By assumption, Forge_1 produces a forged certificate with a *not negligible probability*, and thus the same holds true for Forge_2 .

It remains to replace all references to the private session key ks_{priv} with queries to the signing oracle $S[ks_{priv}]$ to obtain a forging procedure Forge_3 which matches Definition 8 of an adaptive chosen message attack:

Proof (of Theorem 2): We apply Lemma 3 to Forge_1 as defined in Section 5.4 to obtain Forge_2 which produces forged certificates with not negligible probability by Corollary 1. Below, we show the transformed procedure Forge_2 , which uses ks_{priv} only to generate signatures.

F1₂ Key and Random Seed Generation

Simulate the session initialization phase:

- Generate a sequence $\langle kr_{priv}^1, kr_{pub}^1 \rangle, \dots, \langle kr_{priv}^t, kr_{pub}^t \rangle$ of key pairs.
- Generate a sequence R_1, \dots, R_t of random seeds.
- Generate a master key pair $\langle km_{priv}, km_{pub} \rangle$.
- Set round = 0.
- Initialize state with km_{pub} , ks_{pub} , $\text{cert}_{\text{Compiler}}$, and $\text{cert}_{\text{Verifier}}$.

F2₂ Source Code Computation

Simulate Step C1 with a call

$$\text{source} = \overline{\text{MSup_gen}}(\text{state}, \begin{array}{l} kr_{pub}^1, \dots, kr_{pub}^{\text{round}}, \\ kr_{pub}^1(R_1), \dots, kr_{pub}^{\text{round}}(R_{\text{round}}), \\ kr_{priv}^1, \dots, kr_{priv}^{\text{round}} \end{array}).$$

F3₂ Source Code Verification

Simulate Step C2 by computing

- $\langle \log_{\text{Sup}}, \log_{\text{Cus}} \rangle = \text{Verifier}(\text{source})$ and
- $\text{exec} = \text{Compiler}(\text{source})$,
- incrementing round by 1, and

- computing the certificate
 $\text{cert} = \text{csign}(ks_{priv}, \langle \log_{Cus}, \text{exec} \rangle, R_{\text{round}}).$

F4₂ Secrecy Validation (Forge Certificate)

Simulate Step C3, i.e., make a call

$$\text{result} = \overline{\text{MSup_val}}(\text{state}, \text{cert}, \\ kr_{pub}^1, \dots, kr_{pub}^{\text{round}}, \\ kr_{pub}^1(R_1), \dots, kr_{pub}^{\text{round}}(R_{\text{round}}), \\ kr_{priv}^1, \dots, kr_{priv}^{\text{round}}).$$

Depending on result, the execution proceeds with

- Step F5₂, if result indicates to continue.
- Step F2₂, if result indicates to start over again.
- an **erroneous abort**, if result indicates that a secrecy violation has been detected.

F5₂ Output Result

Output the pair $\langle \text{exec}, \log_{Cus} \rangle$ and cert as indicated by result.

Based on Forge₂, we build the attack procedure Forge₃ with two further modifications such that they do not change the not negligible success probability of the overall procedure.

- First, in Step F1₂, we drop the precomputation of the random seeds R_1, \dots, R_t .
- Second, in Step F3₂, we rely on the signing oracle $S[ks_{priv}]$ to generate the certificate cert, and obtain with cextract the random seed that has been used by the oracle to compute the last certificate.

The success probability remains the same, since $\overline{\text{MSup_gen}}$ and $\overline{\text{MSup_val}}$ cannot detect the modifications, and since the random seeds R_1, \dots, R_t are generated in both cases according to the uniform distribution. Below, we show the updated Steps F1₃ and F3₃, while the other ones remain the same.

F1₃ Key and Random Seed Generation

Simulate the session initialization phase:

- Generate a sequence
 $\langle kr_{priv}^1, kr_{pub}^1 \rangle, \dots, \langle kr_{priv}^t, kr_{pub}^t \rangle$ of key pairs.
- Generate a master key pair $\langle km_{priv}, km_{pub} \rangle$.
- Set round = 0.
- Initialize state with km_{pub} , ks_{pub} , $\text{cert}_{\text{Compiler}}$, and $\text{cert}_{\text{Verifier}}$.

F3₃ Source Code Verification

Simulate Step C2 by computing

- $\langle \log_{Sup}, \log_{Cus} \rangle = \text{Verifier}(\text{source})$ and
- $\text{exec} = \text{Compiler}(\text{source})$,
- incrementing round by 1,
- computing the certificate with the signing oracle $\text{cert} = S[ks_{priv}](\langle \log_{Cus}, \text{exec} \rangle)$, and
- extract the random seed $R_{\text{round}} = \text{cextract}(\text{cert})$ from the certificate cert.

Since Forge₂ has a not negligible chance to forge certificates, and since the output of Forge₃ is not changed by the last two changes, Forge₃ has a not negligible chance to forge certificates as well. Moreover, Forge₃ accesses the private key ks_{priv} only in terms of the signing oracle $S[ks_{priv}]$, and runs in probabilistic polynomial time.

In other words, Forge_3 is a successful adaptive chosen message attack, as defined in Definition 8. This is a contradiction and concludes the proof. ■

6. CONCLUSION

IP boundaries impose an obstacle in the dissemination and application of verification techniques. In the commonly considered verification scenario, a relationship of mutual trust between the software author and the verification engineer is presumed: First, the verification engineer believes that the sources provided by the software author have been indeed used to produce the final binary, and second, the software author expects the verification engineer to respect its IP rights on the provided sources.

But in an industrial context, violated IP rights and forged verification verdicts entail enormous monetary damages and henceforth a mutual trust relationship is insufficient as protection against such misconduct. We identified two security properties which are essential for any security solution facilitating 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 which is not already directly encoded within the binary and the verdict.

Taking this situation as starting point, we introduced the *amanat protocol* as a solution which satisfies both, conformance as well as secrecy.

Subsequently, we proved the secrecy of the protocol in an intuitive and cryptographically unconditional manner. This is important as to provide a simple argument which asserts the well-protection of the involved IP in a manner which is convincing to engineering and management staff of a code supplying company. In case of conformance, the proof required a much more technical approach since abstract reasoning on the protocol (e.g., following the Dolev-Yao style) is insufficient to establish conformance.

We also envision wider applications of our protocol: First, we consider deep supply chains in the B2B setting where the final code consumer wants to ensure conformance while facing a possibly maliciously colluding group of chained suppliers. Second, we consider 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 these scenarios will be part of future work.

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 für 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.
- Bullseye. BullseyeCoverage 7.11.15. <http://www.bullseye.com/>.
- 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. 2007. Verification Across Intellectual Property Boundaries. In *Proc. 19th International Conference on Computer Aided Verification (CAV '07)*. 82–94.
- CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D. X., AND BRYANT, R. E. 2005. Semantics-aware malware detection. In *Proc. 26th IEEE Symposium on Security and Privacy (S&P '05)*. 32–46.
- 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.
- CMeter. CoverageMeter 5.0.3. <http://www.coveragemeter.com/>.
- COLIN, S. AND MARIANI, L. 2005. *Model-based Testing of Reactive Systems*. Lecture Notes in Computer Science, 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.
- DANNENBERG, J. AND KLEINHANS, C. 2004. The Coming Age of Collaboration in the Automotive Industry. *Mercer Management Journal* 18, 88–94.
- DARWIN, I. F. 1986. *Checking C programs with lint*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.
- 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.
- FERDINAND, C., HECKMANN, R., AND WILHELM, R. 2004. Analyzing the worst-case execution time by abstract interpretation of executable code. In *Proc. Automotive Software Workshop San Diego (ASWSD '04)*. 1–14.
- 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.
- HOLZER, A., TAUTSCHNIG, M., SCHALLHART, C., AND VEITH, H. 2008. FSHELL: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *Proc. 20th International Conference on Computer Aided Verification (CAV '08)*. Lecture Notes in Computer Science (LNCS), vol. 5123. 209–213.

- HOLZER, A., TAUTSCHNIG, M., SCHALLHART, C., AND VEITH, H. 2010. How did you specify your test suite ? In *Automated Software Engineering (ASE'10)*. accepted for publication.
- KINDER, J., KATZENBEISSER, S., SCHALLHART, C., AND VEITH, H. 2008. Proactive Detection of Computer Worms Using Model Checking. *IEEE Transactions on Dependable and Secure Computing (TDSC)*. <http://doi.ieeecomputersociety.org/10.1109/TDSC.2008.74>.
- KINDER, J. AND VEITH, H. Precise static analysis of untrusted driver binaries. In *Proc. 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*. Accepted for publication.
- 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), 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.
- WENZEL, I., KIRNER, R., RIEDER, B., AND PUSCHNER, P. P. 2005. Measurement-Based Worst-Case Execution Time Analysis. In *Proc. 3rd Software Technologies for Embedded and Ubiquitous Systems (SEUS '05)*. 7–10.
- WIEDIJK, F., Ed. 2006. *The Seventeen Provers of the World*. Lecture Notes in Computer Science, vol. 3600.