Trust model in presence of trusted hardware

Jan Cappaert, Thomas Herlea, Dries Schellekens, Brecht Wyseur

March 20, 2007

Abstract
In this paper, the trust model for RE-TRUST WP3 is discussed.

1 Introduction

2 Trusted components
These are the components we trust:

2.1 Trusted execution environment
An adversary has total control over the untrusted client platform. A malicious user or program (i.e., malware) are assumed to have full privilege access to the system, in their attempts to attack the original program $P$ and additional software $M$ monitoring the integrity of $P$.

2.1.1 Trusted server
The trusted server is not controlled by the adversary. A simple solution to circumvent the remote entrusting problem is server side execution of the program $P$. This however only protects the integrity and confidentiality of the software functionality (i.e., program code), but not the input and output data; for instance, key loggers will still be able to log passwords and other sensitive data. Server side execution also puts a lot of processing load on the trusted server, which is undesirable in certain applications.

2.1.2 Trusted hardware
The trusted hardware can be trusted to perform computations on data, in such a way that the attacker does not learn anything about the data processed and/or the operations executed. The computational power and storage capacity of the secure hardware are limited and thus running the full functionality of the original program $P$ is impossible; again I/O behavior (i.e., communication between untrusted platform and trusted hardware) can always be observed and manipulated by an attacker.
2.1.3 Software splitting

Because execution of the original software $P$ on either the server, or the trusted hardware is undesirable, maybe even impossible, one could opt to only execute security sensitive functionality in a trusted execution environment.

Approaches have been proposed to determine how the split software in open and hidden components, that need to execute on the untrusted environment and trusted environment respectively [24]. In some highly reactive application, such as online games, splitting the execution of $P$ between the untrusted client and trusted server is unacceptable if the interactions with the remote server are synchronous (i.e., the application blocks waiting for a response from the server). Virtual leashing is proposed as an asynchronous solution to overcome this potential problem [11].

2.2 Protecting data

Various cryptographic primitives exist to protect the integrity and/or confidentiality of data. Cryptology has historically focussed on protecting data during transmission. In the white-box security model, where an attacker has full control over the execution environment, traditional cryptographic algorithms are not so strong; e.g., it is rather straightforward to extract an encryption key in computer memory [21] or to deduce a key from cache timing information.

2.2.1 White-box cryptography

White-box cryptography [7, 8, 15] covers a set of techniques to implement a block cipher in order to obstruct the extraction of the embedded secret key. The main idea is to implement the block cipher as a network of lookup tables. All the operations such as the key addition are embedded in the lookup tables, which are then randomized to obfuscate their behavior. Typically, external encodings are applied to increase the level of security, transforming the block cipher $E_k$ into an encoded implementation $G \circ E_k \circ F^{-1}$. Research in this topic is still ongoing, as several cryptanalysis have been presented [4, 12, 23] .

2.2.2 Observable cryptography

Physical observable cryptography is a new methodology mainly used for public key primitives. This research however is still in an early stage.

2.2.3 Computing with encrypted data

The use of privacy homomorphisms as a technique to process encrypted is quite old [1, 18]. Homomorphic functions are functions that have a relation between operations in the (remote) encrypted domain and the local domain, i.e., there is a relation $y = f(x) \rightarrow E(y) = f'(E(x))$. Research is ongoing to achieve more complicated processing in the encrypted domain.
2.3 Protecting software

Several techniques address these problems and try to create self-protecting software [10, 17]. For example, code obfuscation transforms code while preserving its functionality such that analysis becomes hard, expensive and time consuming. In the meantime, related techniques are being developed to protect against software tampering. These mechanisms typically also protect the data software is handling to a certain extent.

2.3.1 Software obfuscation

Code obfuscation to slow down the adversary in attempts to analyze and subsequently tamper software.

Code encryption...
Self modifying code as way to obfuscate software...
[9, 16, 22]
Hardware/software co-obfuscation?

2.3.2 Tamper resistant software

As code obfuscation aims to thwart code analysis, self-checking code tries to protect against tampering. Protecting code against tampering can be considered as the problem of data authenticity, where in this context ‘data’ refers to the program code. Aucsmith [3] was the first to propose a scheme to implement tamper-resistant software. Chang et al. [5] proposed a scheme based on software guards. Their protection scheme relies on a complex network of software guards which mutually verify each other’s integrity and that of the program’s critical sections. The security of the scheme relies partially on hiding the obfuscated guard code and the complexity of the guard network. Horne et al. [13] elaborated on the same idea and proposed ‘testers’, small hashing functions that verify the program at runtime. Other related research is oblivious hashing [6] which interweaves hashing instructions with program instructions and which is able to prove whether a program operated correctly or not.

2.3.3 Software watermarking

2.3.4 Software interlocking

2.3.5 Software replacement

[14]
Different functionality.

2.3.6 Software diversity

Code diversity [2] to produce substantially different implementations of the same software functionality.
2.3.7 Computing with encrypted functions

Encrypted functions that can be executed without prior decryption have been described in [19, 20], and are often also referred to as function hiding. Sander et al. describe how encrypted programs can be used to achieve protection of algorithms against disclosure and can give way to surprisingly solutions for seemingly unsolvable problems of software protection. The key point is to encrypt functions such that they remain executable, but provide encrypted results $E(y) = E(f)(x)$.

References


