IDIOSYNCRATIC SIGNATURES FOR AUTHENTICATED EXECUTION
The TrustedFlow™ Protocol and its Application to TCP

Mario Baldi
Computer Engineering Department
Torino Polytechnic
Torino, Italy
mario.baldi@polito.it

Yoram Ofek
Synchrodyne Networks, Inc.
New York, NY
ofek@synchrodyne.com

Moti Yung
Computer Science Department
Columbia University
New York, NY
moti@cs.columbia.edu

ABSTRACT
Assuring that a given code is faithfully executed with defined parameters and constraints is an open problem, which is especially important in the context of computing over communications networks. This work presents TrustedFlow™, a software solution to the problem of remotely authenticating code during execution, which aims at assuring that the software is not changed prior to and during execution. A flow of idiosyncratic signatures is continuously generated and associated to transmitted data by a secret function that is hidden (e.g., obfuscated) in the software and whose execution is subordinated to the proper execution of the software being authenticated. The flow of signatures is validated by a remote component.

KEY WORDS
Trusted computing; trusted code execution; trusted communication software execution; trusted communication middleware; trusted network applications.

1 Introduction
Software, especially in the context of data networks, suffers from some inherent problems. These include modifications by an either malicious or inadvertent user, malware distribution (e.g., viruses and "Trojan horses"), and the use of malicious software remotely for penetration, intrusion, denial-of-service (DoS), and distributed DoS (DDoS). For example, a rogue user may manipulate the code of a given protocol (such as TCP) and gain an unfair advantage in using network bandwidth.

TrustedFlow™ is a software solution aimed at overcoming these problems and at assuring (in many typical scenarios) that operations are executed by a trusted software source. In current software systems many reactive techniques that try to prevent malware attacks are employed, but are mostly after-the-fact and inaccurate in that they adversely affect innocent users.

The TrustedFlow™ protocol is different and more effective. The solution is based on continuous authentication, ensuring at run-time that the correct software has been employed. Continuous authentication is achieved by a continuous flow of idiosyncratic signatures that are constantly being generated and emanated during execution. This method guarantees that the correct software modules are used at run time and is called TrustedFlow™ because the authenticity of the executed software and its configuration parameters can be trusted.

The TrustedFlow™ protocol has not been designed with a specific application in mind and can be used in various computing and networking scenarios, either by itself or together with other existing security solutions. After having presented some previous work related to the TrustedFlow™ protocol in Section 2 and having described the TrustedFlow™ basic principles in Section 3, this paper presents its application for authenticating the execution of the transmission control protocol (TCP) software (see Section 4). Such application is not intended as a rationale and motivation for the TrustedFlow™ protocol. Rather, being a basic tool to ensure trusted code execution, the TrustedFlow™ protocol can be used in various contexts and enables a plethora of new applications and services.

The TrustedFlow™ protocol is an *add-on software protection component* intended to be included within other protocols, such as those, for example, of distributed computation (e.g., grid computing), traffic generation (e.g., TCP), and management (e,g., SNMP). In essense, the TrustedFlow™ protocol provides run-time continuous (multi-factor) authentication, certifying the authenticity of software modules that were used to compute, generate, and send messages. As such, it becomes evident that the TrustedFlow™ protocol has broad applications in both networking and computing for military and commercial environments. Moreover, in comparison with other network and computer security solutions, the TrustedFlow™ protocol minimizes the number of operations per data packet.

Possible applications of the TrustedFlow™ protocol include: (1) DoS and intrusion avoidance assuring trusted clients, (2) VPN add-on assuring correct software execution (malware free), (3) protecting client applications (from clients) in content handling programs (digital right management - DRM), (4) protecting server applications from misbehaving client programs, such as: malware SSL, malware IPSec, (5) protecting (Java) applets in Peer-2-Peer collaborative computing, (6) protecting the routing infrastructure - avoiding misuse of the various routing protocols, (7) trusted network management, and (8) trusted traffic conditioning, i.e., remote verification of compliance to a traffic profile.

2 Related Work

The TrustedFlow™ protocol complements many of the current enhancements for secure computing and networking protocols, since (to the best of our knowledge) no other authentication method certifies the software continuously during run-time by emanating idiosyncretic signatures. In other words, while other approaches provide privacy and authentication protecting from the attacks of a man in the middle, TrustedFlow™ protects from the attack of *a man at the edge*. The TrustedFlow™ protocol has broad *synergistic implications* on various computing and networking protection means.

2.1 Synergy: Cryptographic Authentication

Cryptographic methods for data authentication, e.g., digital signature and message authentication code (MAC) are used for authenticating data flows — e.g., IPsec , TLS , and SSL . They can be combined with our protocol for data authentication and signing, which is not the goal of the TrustedFlow™ protocol. The combined authentication will assure the data content and the software code that generated it.

2.2 Synergy with Biometric Signature

Human generated idiosyncratic signature is typically associated with biometrics. Biometric signatures generally use biometric data (such as a fingerprint image, a voiceprint recording, or some other replica of a physical feature) in order to identify people. In our context, biometric signatures are used to verify the authenticity of computing and networking users. Although biometric signatures are continuously improving and are generally quite reliable, when the signature information is communicated through a computer over a network, the *software protocol* that is used can be subject to intrusion in various ways (including "Trojan horses"). For example, an attacker could simply bypass the sensor that is feeding biometric data. In order to solve this biometric signature vulnerability hardware is often used, such as tamper-resistance packaging, power interruption circuitry, and a computing hardware module that continuously monitors the biometric system's software to ensure the integrity of the overall operation.

The TrustedFlow™ protocol can be a highly effective measure used to ensure the integrity of the biometric signature together with the software modules used to communicate the information among computers over the network. More specifically, using the TrustedFlow™ protocol would extend the biometric signature to include an idiosyncratic signature of the computing and communicating software modules. The combined biometric and executed software signatures are performed by the computing device that receives the information and requires the user to verify its authenticity. The advantage of this novel solution is that there is *no need for any special hardware*, and consequently can be easily deployed on any system.

2.3 Synergy with trusted computing

Two notable complementary activities will further enhance the TrustedFlow™ protocol by ensuring that the computing platform as a whole is to be trusted (see [2]):

- TCPA (trusted computing platform alliance) is an industry working group that is focusing on improving trust and security on computing platforms [2]. The goal of the TCPA specifications is to enhance hardware and operating system based trusted computing platform that implements trust into client, server, networking, and communications platforms; and
- Next-Generation Secure Computing Base (NGSCB), formerly codenamed Palladium, is an evolutionary set of features for Microsoft Windows. Palladium will provide individuals and groups of users greater data security, personal privacy, and system integrity. In addition, Palladium will offer significant new benefits for network security and content protection.

Both TCPA and Palladium are *comprehensive hardware and software solutions* that require special hardware, while our software solution does not require any special hardware. On the other hand, once the trusted computing platforms are available, the security provided by the TrustedFlow™ protocol will be further increased and it can take advantage of these platforms, enhancing them with its remote assurance function.

3 TrustedFlow™ Protocol Principles

Our solution has two basic components. The first is the preparation and manipulation of the software code at the source. This is needed to assure that the idiosyncratic signature generation is bound in an inseparable manner to the actual functional code. The second component is the protocol generation and checking aspect of the solution during run-time.

The first component (discussed in Section 3.2) generates a "*combined module*" to be executed at run time:

- **Interlocking** is the general term we use to describe a combining of original executable software modules (original function) together with an idiosyncratic (pseudo-random) signature generator (which is used to generate an *unpredictable flow of tags*) forming a "combined module."
- **Program Hiding** The "combined module" is prepared in such a way as to ensure that reverse engineering is difficult enough so that it is functionally infeasible.

During run-time, the following two components define the TrustedFlow™ protocol:

- A **Trusted Flow Generator** (**TFG**) (Figure 1 and Figure 2) executes the "combined module", which constantly outputs its results (such as messages) together with a flow of tags, constituting the continuous *idiosyncratic signature* of the "combined module's" *run-time execution*.
- A **Trusted Tag Checker** (**TTC**) (Figure 1 and Figure 2) is used to remotely authenticate (verify) the flow of idiosyncratic signatures that forms the continuous

idiosyncratic signature emanated from the "combined module", thus assuring that the correct "combined module" was executed in run-time.

3.1 Basic Principles

The TrustedFlow™ protocol, as shown in Figure 1, is based on a Trusted Flow Generator (TFG) in the trusted-to-be programs and a Trusted Tag Checker (TTC) function on another computer or as part of some network interface (e.g., firewall, gateway). The TFG contains a hidden (e.g., obfuscated [1]) program part that generates a pseudo-random sequence of n-bit tags (idiosyncratic signature). The n-bit tags (where n is small - typically only 1 bit) are included in the sequence of messages (e.g., inside data packet headers) that are sent from a first computer through the network to a second computer. At the second computer, the validity of the pseudo-random sequence of n-bit tags is checked and verified by the TTC. Sending a valid pseudo-random sequence of n-bit tags verifies that the first computer has used the appropriate software (programs and parameters). Consequently, the second computer accepts and/or forwards only data packets from well-behaved sources.

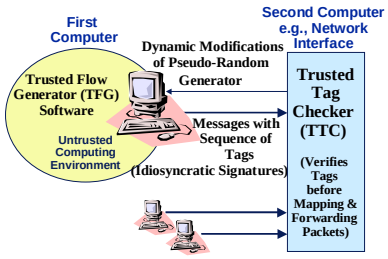

Figure 1: TrustedFlow™ architecture

A "**reverse**" TrustedFlow™ protocol, as shown in Figure 2, is a variant of the TrustedFlow™ protocol, such that, the TTC function is not in the network interface but in the end device. The "**reverse**" TrustedFlow™ can be used, for example, for protecting mobile/wireless devices. Such devices typically feature limited computing and storage capabilities. In this case, the mobile device contains the TTC functionality in order to off-load in a trusted manner some computations, such as anti-virus programs, to a networked computer, which contains the TFG functionality. The rationale is that the base station has vast computing resources, while mobile devices are limited in computing and storage capabilities.

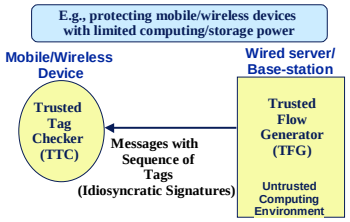

Figure 2: Reverse TrustedFlow™ architecture

As mentioned, the TFG contains a hidden program that is obfuscated into the program used for generating and sending messages. More specifically, the TFG contains the program for sending messages, which contains an obfuscated secret part that generates a pseudo-random sequence of n-bit tags (idiosyncratic signature). Only the TTC is able to locally generate the sequence of n-bit tags. The TTC then compares the locally generated sequence with the received sequence. A successful comparison verifies that the legitimate program was used to generate and send the messages. Next, we briefly discuss what is obfuscation, which is one of the key components of the TrustedFlow™ protocol.

3.2 Program Hiding with Secure Interlocking

In general, the TrustedFlow™ protocol assures that a "combined module" execution is performed by a computing system. Part of this "combined module" is the original computing functionality (or plain program, see Figure 3) whereas some other part of this "combined module" is a signal generation method, which produces unpredictable signals (by, for example, a cryptographically strong pseudo random generator). The mechanisms make sure to **interlock** the parts into the "combined module" - see Figure 3.

The **interlock** means that *all parts must be performed at the same time*. The operation part, which is factored into the "combined module," is trusted (and is associated with an execution of specific computations (e.g., a network access method), operation limitations (such as transmission rate, number of times before renewal of precondition for next operation, etc.). The signal generation produces unpredictable signals as its "idiosyncratic signature." The checking is done merely by being able to reproduce and check the signature that could have only be generated at the interlocked program (due to the strength of the interlocking and the cryptographic unpredictability of pseudorandom signals). If the signature verification passes, it means that the other (operation) part of the interlocked module was performed as well. Thus, it was performed subject to the associated limitation, namely as a trusted one.

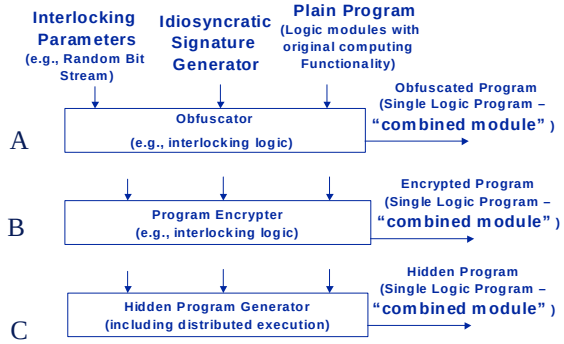

Figure 3: Various possible methods for constructing the secure "combined module"

As shown in Figure 3, there are various possible methods for constructing the "combined module" as a single logic program. The system for providing the "combined module" gets as inputs: (1) random bit stream as an

interlocking parameter and (2) plain program consisting of logic modules **(**as shown in Figure 3**).** It performs the integration of the logic modules using three different methods for secure integration in a single logic program – "combined module":

- An "obfuscator" (Figure 3A), and/or
- A "program encrypter" (Figure 3B), and/or
- A "hidden program generator" (Figure 3C).

The first method of software obfuscation (Figure 3A), relies solely on software methods. It involves transforming the module into a functionally equivalent but hard to understand module. Obfuscation methods have been studied and even automated [1][3][4]. They rely on transforming the code in various ways so that the result is a scrambled version of the original code. While, it is understood that with a lot of reverse engineering efforts, one can de-obfuscate a program, the technology is such that the reverse engineer needs a lot of effort. Since the modules can be obfuscated every so often (i.e., dynamically) and with new signature parameters, the task of reverse engineering the module can be made hard and non cost-effective. Obfuscation methods, also known as "tamper proof software" have been studied in recent years in various software technologies [2], and several companies offer such services. Let us review some aspect of obfuscation methods.

3.2.1 Obfuscation

A program is transformed using an automated method that is formally defined as:

Obfuscation*: an obfuscator O is a probabilistic "compiler" that takes as input a program (or circuit) P and produces a new program O(P) that has the same functionality as P, yet is "unintelligible" in some sense.*

Most of the obfuscator applications are based on an interpretation of the "unintelligibility" condition in obfuscation as meaning that O(P) is a "virtual black box." Code obfuscation is similar to code optimization, however, obfuscation maximizes obscurity while minimizing execution time, whereas optimization only minimizes execution time.

3.2.2 Dynamic modification of obfuscated codes and parameters

Currently, obfuscated codes are used primarily for content protection programs and programs that manage content locally, and not for interlocking two programs securely for remote signaling (e.g., by using secure sequence of n-bit tags). A novelty of TrustedFlow™ protocol is the use of obfuscation (or other software-hiding methods) to "interlock":

1. The known performance characteristics of the packet generation portion of the software that affects the entire network and servers, together with
2. The unknown unpredictable individualized sequence of n-bit tag (pseudo-random) generation software used for secure signaling.

This in turn enables the codification and verification of various other aspects (such as, type of check, performance parameters, etc.) by using *dynamic modification of the pseudo-random generator* in the TFG, as was shown in Figure 1, in order to dynamically manage and refresh the obfuscated part of TFG. This serves to increase hardness of de-obfuscation and making de-obfuscation functionally infeasible.

3.2.3 Other hiding methods

There are other possible methods for program hiding. The second method of program encrypter (see Figure 3B), relies on having an end-to-end secure authenticate channel between the program generator and the place where the program is to be run. It may be that the program is then decrypted and executed in a "tamper proof" part of the architecture. A particularly attractive method for implementing this may be the TCPA (Trusted Computing Platform Alliance) architecture [2], led by major IT companies. The architecture enables portions of the software to be executed without the user controlling it. Originally, it may have been motivated by Digital Right Management of content distribution, but it can be used for execution of software that we do not want to allow the local user an ability to alter.

The third method (Figure 3C) may embed the combined module as is in a co-processor that is protected or hide part of the execution is protected smart cards. This method may assume that the function of the functional module may be or should be performed at a remote location or a separate card (e.g. a network controller).

The result of the three methods is the formation of "combined modules", which make it practically impossible to learn how the unpredictable idiosyncratic signature is generated. Note again that the objective is not necessarily to hide the original computing functionality (or plain program), but rather to hide a "secret signature generating function" within the original computing functionality.

The system for a secure "combined module" integrates the operational component and the signature component into a virtually single logic module. This module hides certain rules of execution, thus providing that the mechanism is assured to interlock the idiosyncratic signature generation and the rules of execution. In particular, the system is run wherein one of the software logic modules provides rules of transmission for accessing the network. These rules of transmission may compute certain limitations and conditions on the operation of the protocol: performance characteristics, access characteristics, transmission limitations, transmission rates, window sizes, port numbers, IP addresses, network addresses, quotas, renewable quotas, packet structure limitations, schedule. Indeed, it is possible to combine rules as well to assure that a number of execution rules are being followed for various transmissions distributed computation operations. The result is that a mere check of signature validity assures that the operational module with the constraints and limitation was followed by the software. Thus, many assurances are encompassed within the validity of the signature, due to the notion of "interlocked execution."

4 Trusted TCP: an Example of TrustedFlow™ Deployment

This section describes a sample deployment of the TrustedFlow™ protocol in data networks. In particular, the paper provides a high level description of how to perform secure interlocking of a TFG in the transmission control protocol (TCP) software that is responsible for flow and congestion control over the Internet.

The TrustedFlow™ protocol has not been designed with this specific application in mind; hence, the content of this section not intended as a rationale and motivation for the TrustedFlow™ protocol, but rather as an example of its deployment. The reason for addressing secure interlocking of TCP as a TrustedFlow™ application is twofold. First, being the TCP code publicly available and its functional specification well known, it is suitable to be used as a case study. Second, as explained in Section 4.1 below, modified TCP software can cause unfair sharing of network resources — if deployed on a small scale — or even a global failure of the Internet — if widely deployed.

4.1 TCP Principles of Operation and Motivation for Secure Interlocking

TCP includes a number of operational rules for error, flow, and congestion control. A sliding window mechanism limits the amount of data sent into the network before receiving confirmation of its reception (a.k.a. acknowledgment). TCP congestion control aims at recovering from congestion, or avoiding it in the first place, by reducing the transmission rate. The sender uses segment loss events to infer congestion in any routers on the path of its TCP connection. A segment loss event is identified by either the expiration of the time-out associated to the lost segment, or a duplicated acknowledgement received for a previous segment. The transmission rate is changed by modifying the size of a transmission window according to an algorithm called AIMD (additive increase, multiplicative decrease). This algorithm aims at trying to achieve maximum utilization of network resources by slowly increasing — additive increase — the window size during periods of low congestion. As congestion is detected, the sender is supposed to react immediately by significantly reducing its transmission rate, i.e., the size of its transmission window — multiplicative decrease.

If all TCP senders using a network resource, such as a link, behave according to the defined TCP flow and congestion control algorithm, they all share (more or less equally) the network resource. On the contrary, if a TCP source is misbehaving — for example, not reducing its transmission window size — it could grab a large fraction of the bandwidth and shut up well-behaving ones. Thus, misbehaving TCP senders can compromise the network performance as perceived by the users deploying well-behaving TCP code.

The gravity of the situation arising by a large number of such misbehaving TCP senders can be fully understood by looking back at the motivations for the above mentioned congestion control algorithms. The original TCP specification did not include such features, i.e., the protocol operation was based on a fixed size congestion window. This was the cause of the severe and irreversible congestion that the Arpanet experienced in October 1986: the average throughput on the 32 kb/s links constituting the network was down to 40 b/s. Heavy traffic brought to the congestion of links, buffers feeding them overflowed, and packets were discarded. Consequently, TCP senders, not receiving acknowledgements for lost transmitted packets, timed-out and retransmitted a whole window-worth of packets (including some possibly received out-of-order from their destination), thus perpetrating the congestion. The only way out of this congested state had been closing a large number of TCP connections by terminating the applications that opened them.

The introduction of congestion control based on varying TCP's transmission window size has eliminated the risk of such an irreversible congestion state. However, the conditions that triggered the irreversible congestion experienced in 1986 would be reproduced if a very large number of users deployed TCP code modified to achieve larger throughput by avoiding properly reducing the transmission window. Since the consequence of such an irreversible congestion is the impossibility of communicating through the whole Internet, such a modified TCP code could be distributed as a way of carrying on as a global, distributed denial of service attack. Achieving widespread deployment of the modified code could not even be very difficult: for example, the modified TCP code for the main operating systems could be made publicly available for download and advertised as yet another simple fix to improve the performance of file transfers and web page downloads (many such "fixes" are currently being offered). This would trigger a time bomb that would approach its detonation as more and more Internet users install the "improved" TCP code. The time of the "Internet explosion" would be unpredictable and the phenomenon completely unexpected.

Interlocking a TFG with the TCP software allows the network to verify that a TCP sender is executing a proper TCP implementation and hence it is well behaving. A Trusted Tag Checker (TTC) at the network boundary, as shown in Figure 1, can identify TCP segments not generated by Trusted TCP Code — that plays the role of the TFG in the architecture depicted in Figure 1 — and apply to them a pre-defined handling policy, such as discarding the frame or sending it with lower priority.

4.2 Realization of Secure Interlocking for Trusted TCP

This section describes a possible TrustedFlow™ protocol realization that allows the network to verify the proper operation of a TCP implementation, which is labeled Trusted TCP Code. Figure 4 is a functional block diagram of the high layers of a TCP/IP protocol stack used by various applications to transmit and receive data units to and from a multiple other applications through the socket interface layer.

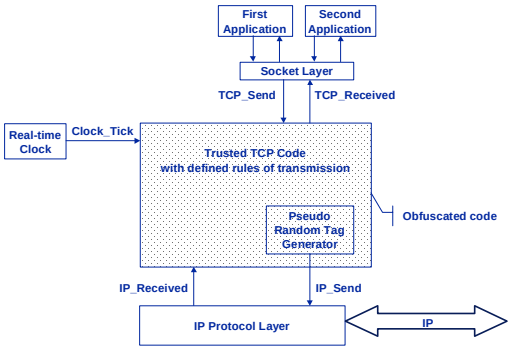

Figure 4: Example: Trusted TCP layer (a TFG implementation)

The Trusted TCP Code includes a pseudo-random generator of security information — *n-bit tags* — to be associated to the TCP segments being transmitted. The security tag associated to a TCP segment have to objective of certifying that the TCP segment was generated by Trusted TCP Code, i.e., that the TCP transmitter that originated a TCP segment operates in compliance with the rules of transmission — i.e., flow and congestion control — specified by a given TCP standard (e.g., TCP Reno).

Figure 5 describes the operations performed by the Trusted TCP Code for the transmission of a TCP segment. Besides an n-bit tag, a *Security Tag Serial Number (STSN)* is associated to each TCP segment prepared for transmission. The n-bit tag is obtained from a pre-computed pseudo-random bit sequence that is known to both, and only, the TFG and the TTC; the STSN identifyes the n-bit tag position within the sequence. The n-bit tag and its STSN are transmitted in the option field within the TCP header. The STSN is essential for a TTC to locate within the pseudo-random bit sequence the n-bit tag contained in inspected TCP segments that were received out of order or after a lost TCP segment.

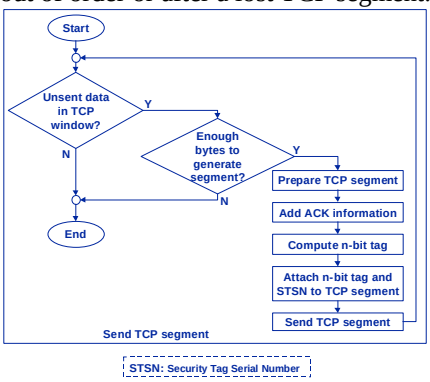

Figure 5: Transmission of a TCP segment

The TTC embedded, for example, in a network appliance at the boundary of a network domain, checks that the n-bit tag associated to each TCP segment has been properly generated, i.e., that the TCP segments were sent by Trusted TCP Code. If the check on the tag reveals that a TCP segment was not properly generated, it is discarded or mapped to a lower priority service through the network domain. In the former case, TCP senders running non trusted code cannot at all use the network services. In the latter case, while non trusted TCP code can transfer data through the network domain, it cannot negatively affect the performance achieved by Trusted TCP Code. In fact, in case of congestion, TCP segments generated by non trusted senders are discarded with higher probability than TCP segments generated by Trusted TCP Code.

The TrustedFlow™ protocol assures proper operation of a TCP sender — and, in general, of the sender of a data stream — not the authenticity or privacy of transmitted data. For applications that require authentication and privacy of transmitted data, a proper authentication and/or privacy protocol, such as IPSec, TSL or SSL, can be used to complement TrustedFlow's capabilities.

5 Conclusions: Creating Trusted System Environment

TrustedFlow™ is a general software tool with wide commercial and military applications. In essence, the TrustedFlow™ protocol creates a combined trusted computing/networking environment by using an idiosyncratic signature — generated by a trusted flow generator (TFG) component operating in an untrusted environment — that is checked by a trusted tag checker (TTC) component operating in a trusted environment (e.g., firewall). TrustedFlow™ complements trusted computing platforms by enabling remote verification that the right software is executed; it constructs a "remote trust" mechanism.

This paper presents the basic principles of the TrustedFlow™, its synergies with other existing network security approaches, and an example of its deployment to authenticate the execution of the transmission control protocol (TCP) code. The TrustedFlow™ protocol has not been designed with this specific application in mind; authentication of TCP software is presented as an easy to understand and meaningful example of its application, not with the aim of providing a rationale and motivation. Rather, being a basic tool to ensure trusted code execution, the TrustedFlow™ protocol can be used in various contexts and enables a plethora of new applications and services.

In comparison with other network and computer security solutions, the TrustedFlow™ protocol has a lower complexity (no need for special hardware), and thus, is less expensive and more scalable. The TrustedFlow™ protocol can be used in various computing and networking scenarios, either by itself or together with other existing security solutions. A prototype of the TrustedFlow™ architecture sponsored by Microsoft Research is under development at Torino Polytechnic.

References

[1] C. Collberg, C. Thomborson and D. Low, "Watermarking, Tamper-Proofing, and Obfuscation-- Tools for Software Protection," IEEE Transactions on Software Engineering, vol. 28, no. 8, 2002.

[2] S. Pearson, B. Balacheff, D. Plaquin, and G. Proudler, "Trusted Computing Platforms: TCPA Technology in Context," Prentice Hall

[3] E. Valdez and M. Yung, "Software DisEngineering: Program Hiding Architecture and Experiments," Information Hiding 1999.

[4] E. Valdez and M. Yung, "SISSECT: DIStribution for SECurity Tool," ISC 2001, pages 125-143, 2001.