# Identification of Source Applications for Enhanced Traffic Analysis and Anomaly Detection

André Zúquete
Dep. of Electronics, Telecommunications and Informatics
IEETA, Univ. of Aveiro
Campus Univ. de Santiago, 3810–193 Aveiro
Email: andre.zuquete@ua.pt

Miguel Rocha
IEETA, Univ. of Aveiro
Campus Univ. de Santiago, 3810–193 Aveiro
Email: miguelrocha@ua.pt

***Abstract*—This article presents an architecture for managing the identification of applications responsible for generating traffic in a network. The identification is to be explored by network auditing systems, which cooperate with surveyed systems to get the relevant information about the source applications. The ultimate goal of the system is to provide network auditors, such as NIDS, enough information about the exact sources of network traffic. This way, auditors are able to detect unauthorized applications or to detect anomalies in the traffic created by known applications, possibly as a consequence of the action of some malware in the source application or host.**

## I. Introduction

Network intrusion detection systems (NIDS) are valuable tools for detecting anomalies in the traffic generated by a network of hosts under surveillance. However, NIDS are not capable of knowing, solely from the traffic they inspect, the application that was responsible for sending it. This information could enable a NIDS to detect abnormal applications' behaviour, in the exact same way as host-based intrusion detection systems (HIDS) or personal firewall systems do. Furthermore, it could enable NIDS to enforce authorization policies regarding applications running in the surveyed hosts.

In a previous work, we have proposed an architecture for tagging traffic with source-related tags [1]. This tags are transmitted in a way that complies with existing IP standards – encapsulated in a new IP header option field. Four different tags are included in this option: Entity Pseudonym (EP), Role Identifier (RoleId), Application Identifier (AppId) and Authenticator (see Fig. 1).

**EP** is a 6-byte pseudonym of a person or a service that is exploring the source host. Pseudonyms are temporary; the same entity may have different pseudonyms on the same machine along the time. A specific network service, the Pseudonym Manager (**PM**), maps EP's to detailed entity-related information. This information is provided only to authenticated and authorized network auditors, such as authorized NIDS. **RoleId** is a 8-byte identifier of the role played by the source entity when running the source application. This field helps to separate the traffic generated by an entity into many different entity-defined roles. The ultimate goal of this field is to facilitate the detection of traffic anomalies by enabling the implementation of a previous, role-based traffic separation strategy. **AppId** is a 8-byte identifier of the application responsible for the packet. Finally, **Authenticator** is a cryptographic packet origin authenticator.

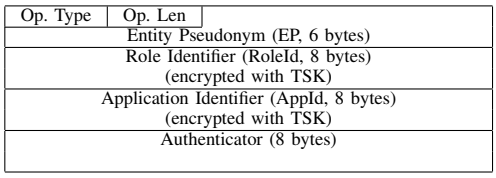


Fig. 1. Structure of the IP option containing the source tags

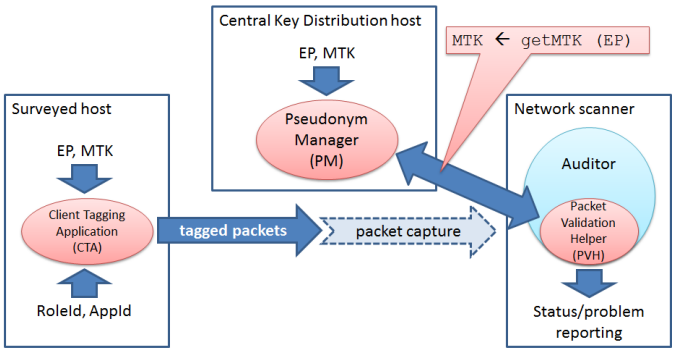


Fig. 2. Packet tagging architecture proposed in [1], where a traffic tagging application on a surveyed host shares a temporary pseudonym (EP) and a related master tagging key (MTK) with a Pseudonym Manager, and an authorized auditing application uses PM for getting the MTK given an EP

For privacy sake, the tags RoleId and AppId are encrypted with a tag secrecy key (**TSK**). The PM maintains a master tagging key (**MTK**) for each current EP, from which TSK is computed. The PM distributes MTK upon request to authorized network auditors, which enables them to compute TSK and observe the cleartext value of RoleId and AppId of captured packets (see Fig. 2).

The architecture described in Fig. 2 assumes a cooperation between surveyed network hosts and network auditors. The absence of this cooperation, or the provisioning of wrong information by the surveyed hosts, will eventually trigger the detection of a generic host anomaly. On the other hand, the provisioning of correct information regarding compromised applications, or unwanted applications, will as well eventually trigger the detection of an anomaly relatively to application-specific network profiles (e.g. abnormal traffic generated by an application infected by a virus) or an anomaly relatively to the set of expected applications running in the surveyed host. Therefore, ultimately we should be able to detect the

activity of malicious or otherwise unauthorized code running in surveyed hosts by using produced traffic and its source as evidence of the presence of that code.

### A. Problem

The space available for options within IP header is limited, and the space occupied by the tag-carrying IP option is already close to the limit. Therefore, tags are very short for including detailed information, namely about the source application. This means that AppId can only be an index of more detailed information about the source application, which should be stored somewhere and provided to authorized auditing applications when needed. This issue was not solved in the original paper describing the packet tagging strategy.

### B. Contribution

This article presents an architecture for managing AppId tags and for establishing bounds between AppId values and detailed information about applications. The architecture is totally distributed, in the sense that there is no central repository for storing the information related with the identification of applications. Furthermore, the information about applications is produced and stored on a needed basis.

Each surveyed host manages a local set of bindings between local applications and their local AppId. Therefore, hosts conforming with our tagging system do not need to interact with any central system for managing AppId's. Network auditors that analyse AppId tags do not also use any central information registry; all their information is mainly collected from captured network packets and from the respective source hosts.

For a proof of concept, we implemented a Linux prototype of the components of the architecture. Namely, we developed an application for binding applications to AppId tags and an application for capturing packets, collect information about their source application from the AppId, and build a registry with links between captured packets and detailed information about their source applications. Performance evaluations of the most frequent and time-critical operation, the fetching of an AppId of an enrolled application in the traffic source host, show that the latency penalty imposed to the packet tagging procedure is acceptable, but has room for improvements.

## II. Architecture

The architecture for managing information about applications responsible for producing network traffic is formed by two main components (see Fig. 3): a registry local to the traffic source (Source Application Registry, SAR), and a registry local to each auditing tool (Auditing Register, AR).

### A. Source Application Registry (SAR)

The SAR is the component that keeps, on each host (operating system), a mapping between applications and AppId's. This mapping is created on a needed basis, not in advance.

Each time an application sends a packet to the network, the packet must be tagged with an AppId, and the packet tagging application gets it from the SAR. The SAR keeps a database of AppId's previously assigned to local applications and checks if the current source application is already registered or not. If yes, then it already has an AppId, which is provided to

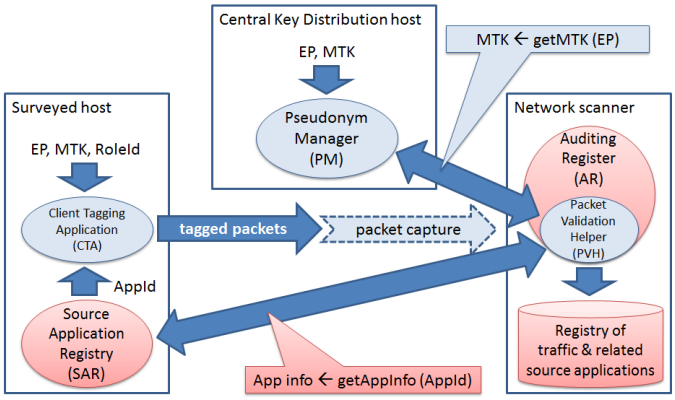


Fig. 3. Evolution of the tagging architecture of Fig 2 with the Source Application Registry (SAR) and the Auditing Register (AR) components

the tagging application. If not, then the SAR allocates a new AppId and enrols the application in the database. Therefore, the database is not build in advance with records of all existing applications, but instead only when needed.

The SAR database maintains, for each AppId, a record with the following fields:

- The name of the package containing the application;
- The version of the package;
- The path name of the application's binary;
- The last modification time of the application's binary;
- A digest (e.g. with SHA-1 [2]) of the application's binary;

The operation of SAR to find an application's AppId, possibly enrolling the application in the database, has the following steps:

1) Given a process identifier (PID), provided by the packet tagging application, the SAR begins by finding the path to the binary of the related process and reads its last modification time.
2) Queries its database, searching for an entry with the path and the observed last modification time.
3) If there is a record with these fields in the database, its AppId is returned.
4) Otherwise, SAR will search for the application's package name and version, computes the binary digest, allocates a new AppId and creates a new database record for the application.

This AppId fetching process is very efficient for getting the AppId of registered applications, being naturally slower for applications when they are absent and are first registered. The modification time of the binary is a simple but yet powerful enough mechanism for detecting the upgrade of applications, since usually such upgrade implies a change in the binary modification time.

*1) Modification time vs. content digest:* An alternative approach for detecting modifications on applications could be to use binaries' digests. This is the approach that is usually followed by personal firewalls, for implementing application-based authorization policies, or by HIDS, for detecting abnormal modifications in the application's code (e.g. in Tripwire [3]). But using digests is much more expensive and the possible advantage is not interesting for us, as explained below.

As a matter of fact, for malware it is easier to deceive SAR when this uses modification time of binaries instead of the digest of binaries' contents. However, we are not interested in detecting malware using evidences of modifications in binaries, but rather to detect malware using evidences of modifications in the traffic generated by applications. Therefore, if an application's binary is tampered by malware, and if that modification is hidden from SAR, the tampering will ultimately, and hopefully, be detected by network auditors that are able to assert the application's normal network behavior.

The digest kept in each application record is maintained for enabling network auditors to detect applications that have the same name, path, and belong to the same exact package, but nevertheless are effectively different at the binary level. Note that this may simply happen because the binaries were build for different hardware architectures, or compiled differently. But it can also be used by network auditors to detect unexpected modifications of the applications' binaries, which may be an evidence of modifications introduced by malware. Therefore, SAR checks the digests on a regular basis, and upon update creates a new record for the application, with a new AppId, and removes the previous record.

*2) AppId allocation and database identification:* The AppId is allocated locally using a simple counter, starting in 1, and growing consecutively up to $2^{64}-1$, the maximum value supported by the packet tagging mechanism. This range for AppId is high enough for preventing its exhaustion forever.

Different hosts will allocate the same AppId for different applications. This is not a problem for network auditors as long as they are able to identify the source database. In other words, we can promote AppId to unique identifiers just by concatenating them to a unique database identifier.

Consequently, each database managed by a SAR instance has a unique identifier (**DbId**). A DbId is a 128-bit value generated randomly each time a database is created by a SAR instance. A 128-bit space is high enough for preventing involuntary DbId collisions. Nevertheless, the unlikely occurrence of a collision between DbId's can be solved in a very simply way just by removing both and creating new ones. Since the whole auditing system using AppId's relies on a cooperation between network auditors and surveyed hosts, this simultaneous reset of DbId's can be triggered using a direct contact with the persons responsible by the involved hosts.

*3) Remote querying:* The SAR has a server interface that allows network auditors to query information about local applications, given their AppId. The service interface has only one function, which receives an AppId and an access authorization credential and returns the DbId and all the data of the record containing that AppId.

The authorization credential prevents unauthorized clients from getting information about the applications registered in SAR. This protection is mainly included for enforcing a need-to-know policy, i.e., SAR servers only provide information to clients that can make good use of it. Access credentials are composed by a fresh, random response encryption key (REK) encrypted with an access key (AK). This AK is a key derived from MTK following the same approach taken in [1]:

$$\mathrm{AK} = \mathrm{KDF}\left(\mathrm{MTK}, \text{“Access key”}\right) \quad (1)$$

where KDF means key derivation function, defined in [4] as an iterative construction based on a pseudo-random function.

MTK must be known by an authorized network auditor, otherwise it cannot have access to TSK and to the cleartext AppId's from captured packets (see Figs. 1 and 2). Therefore, from the SAR's point of view, anyone that also knows MTK would naturally be an authorized network auditor.

The SAR server decrypts the received access credential with AK, uses the result as REK, and encrypts the whole response with it. This way, only legitimate clients will have access to the cleartext of the response.

$$\mathrm{C} \rightarrow \mathrm{SAR} : \{\mathrm{REK},\ \mathrm{AppId}\}_{\mathrm{AK}}$$
$$\mathrm{SAR} \rightarrow \mathrm{C} : \{\mathrm{DbId},\ \mathrm{AppId},\ \mathrm{App.\ information}\}_{\mathrm{REK}}$$

Fig. 4. Query/response dialog between the SAR server and a client C

The SAR client uses also REK for authenticating the response, since it must contain the AppId supplied in the request. Together, the AppId and the REK act as challenges, which the SAR must use properly, with the correct AK, to produce an acceptable response.

### *B. Auditing Register (AR)*

The AR is an application that captures network packets and links them to detailed information about the source application. In other words, it resolves the AppId field in each captured packet to detailed application information provided by a SAR server.

The AR maintains some volatile information in caches to speed up its operation. These caches are used to resolve sets of identifiers to other identifiers, as presented in Table I.

TABLE I
IDENTIFIER MAPPINGS CACHED BY AR TO INCREASE ITS PERFORMANCE

| | |
|---|---|
| EP $\rightarrow$ TSK | Mapping between a packet EP and the key used to encrypt the packet's RoleId and AppId. |
| EP, Src IP $\rightarrow$ DbId | Mapping between a packet EP and source IP address to a DbId of its SAR database |

The operation of AR is the following:

1) Captures a packet and looks for the IP option with the source tags.
2) Resolves EP to TSK, using a local cache (see Table I) or the PM (to get an MTK and, from it, TSK).
3) Decrypt with TSK the IP option field that contains AppId, retrieving this last identifier.
4) Resolves the EP and the source IP address to a DbId using the local cache (see Table I).
5) If this last resolution succeeds, queries a local database for and entry indexed by DbId and AppId.
6) If steps 4 or 5 do not succeed, queries the SAR service on the packet's source host for information regarding its DbId and the packet's AppId.
7) Updates the mappings cache with the information about DbId and updates the local database containing the information about applications registered in all known SAR databases.

The AR only uses remote communications, with the PM or with a SAR service, when strictly required. After some time of operation, and after processing enough packets from surveyed hosts, an ATS database becomes sufficiently populated to reduce contacts with remote SAR services to very sporadic situations (new applications installed, upgrade of existing applications, etc.). The remote interaction with the PM depends mainly on the frequency that the hosts (or the users using them) get a new EP.

## III. Implementation

We implemented our application identification system as a complement of the system implemented in [1]. It was implemented im Linux and we used SQLite to manage databases. The SAR application was implemented in C, while the AR application was implemented in Java.

Our AR was implemented with and extra feature: a graphic interface for presenting the captured packets and the associated source application information. This interface allows a network operator to monitor in real time the applications that are responsible for the current traffic.

Our SAR implementation uses the `/proc` file system to find the path to the binary of an application, given its PID (in file `/proc/[PID]/exe`). For finding the package of a binary, it uses `.list` files stored in `/var/lib/dpkg/info`, which files and their contents need to be searched sequentially until finding a match. Once the package name is known, the application finds the package version by searching in the file `/var/lib/dpkg/available`, where is kept detailed information about all packages. The version is extracted from this file as a text string (e.g. for Ubuntu 10 the current Firefox 3 version is 3.6.24+build2+nobinonly-0ubuntu0.10.04.1).

The SAR implementation is in fact formed by two components: (i) a library component, that provides an AppId given a process PID and updates the database of AppId's if required, and (ii) a server process, that provides a DbId and detailed application information given an AppId. The library is used by the application that tags outbound packets. The SAR server listens for UDP packets from remote clients, namely AR instances. Both components access the same SQLite database. The DbId of the database is stored in itself.

The UDP messages exchanged between a SAR server and an AR are encrypted with AES-128 [5]. For computing a 128-bit AK out of MTK, we used equation 1 and the HMAC-SHA-256 [6], [2] to implement KDF, as suggested in [4]. The result of KDF, a 256-bit digest, is transformed to a 128-bit AES key by folding, with XOR, the high-order 128 bits with the low-order 128-bits.

SAR database has two tables: PKG_TBL stores the name and version of a package; APP_TBL stores the path name, the modification time, the content digest and the index of the package entry in PKG_TBL. The AppId is not stored, it is the index of an entry in APP_TBL. Entries in APP_TBL are in fact never removed when no longer needed, their content is simply filled with zeros.

The consequence of this non-removal policy is that at certain time in the future, after many upgrades and adding/removing many applications, the database can be filled with holes or with useless allocated entries. In this case, a host can decide to compact the database, which implies changing its DbId to a new value and discard any EP currently in use. For network auditors, after this operation a host will be regarded as a new machine, with a new database instance. A problem remains, though, which is the garbage collection of useless information in network auditors regarding databases that "disappeared" upon being compacted (due to its renumeration). The handling of this issue is scheduled for future work.

Our current AR implementation does not store persistently information about captured packets. Information about them is partially stored in volatile memory, just for being presented by the graphic interface. The information stored, for each packet, is composed only by the source and destination IP address, transport protocol, source and destination transport ports, AppId and DbId. The AR uses a single database table to store information returned by SAR servers; that information is indexed by the concatenation DbId with AppId.

## IV. Security Evaluation

The proposed system maintains the privacy guaranties of the original architecture. The remote dialogs between network auditors and surveyed hosts does not reveal any useful information to eavesdroppers, namely the AppId and related information, as they do not know the access key AK nor the response encryption key REK. Past AppId resolution requests can be replayed, but with no other benefit for anyone than performing a denial of service (DoS) attack, because the answer will always be the same.

On the other hand, no one can have access to information maintained by a SAR without knowing the current MTK of its host. Since this key changes each time the host gets bound to a new EP, network auditors must regularly prove their legitimacy, namely each time an host's EP changes. However, attackers can send bogus requests to a SAR, which generate bogus replies (i.e. replies about an AppId unknown to the attacker and encrypted with an REK also unknown to the attacker). Therefore, a SAR is open to DoS attacks using bogus requests, an issue that we need to tackle in the future.

Because AR's use a different, random REK on each request and the AppId of the request must be in the response, replayed responses are detected an discarded. Furthermore, randomly-filled responses spoofed by attackers are unlikely to succeed, since the probability to contain a valid AppId is $2^{-64}$. Therefore, SAR responses to an AR cannot be spoofed by attackers to achieve a benefit other than to create a limited DoS scenario.

## V. Performance Evaluation

Regarding performance, the component that is critical to evaluate is SAR, as it has direct impact in the delay imposed to outbound traffic. However, it is not very relevant to assess the time it takes to create a new database entry, as it has impact on the performance of outbound communications only the first time an application creates outbound traffic. On the other hand, the performance of SAR when retrieving an AppId given a PID is very relevant, as it will happen very often.

For assessing the performance of this last case, we pre-registered in SAR's database different sets of applications existing in one Linux installation, and we measured the

TABLE II
PERFORMANCE IN THE FETCHING OF AN APPID GIVEN AN APPLICATION'S PID WHEN QUERYING 50% OF THE REGISTERED APPLICATIONS

| Registered applications | Registered packages | AppId query performance | | | |
|---|---|---|---|---|---|
| | | min (ms) | avg (ms) | max (ms) | $\sigma$ |
| 100 | 65 | 2.1 | 3.2 | 14.3 | 1.9 |
| 500 | 188 | 5.6 | 13.0 | 362.6 | 36.3 |

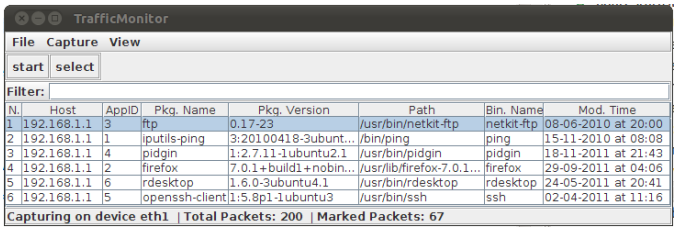


Fig. 5. Example of the graphical interface of AR, showing the 7 source applications responsible for 67 marked packets among 200 captured packets

performance to query a random sample of 50% of them. The results, evaluated on a Linux ArchLinux running on a Intel x86 Core 2 Duo at 2.53 GHz, are presented in Table II. Note that the presented values account the time expended in the entire process of resolving a PID to an AppId.

The performance figures are reasonable, in the sense that they do not increase noticeably (for a human) the latency of outbound packets, but there is room for improving them in future implementations. The current database queries using variable-length strings (path names) are not very efficient, but they are straightforward, as binaries' path names are unique identifiers. Therefore, finding a faster querying strategy for getting an application's AppId is a task for future work.

## VI. EXPERIENCE

We have not yet tested the whole system in a production environment, therefore we do not have yet many experience in the exploitation of the system. Nevertheless, we show an example of the information provided by AR to an operator. In Fig˙ 5 we show its interface after capturing 200 packets, 67 of them tagged and produced by 7 applications of the same host. Currently, we are not yet showing detailed packet information associated with source applications, but we are actively working on it.

## VII. RELATED WORK

There is a vast number of publication of the field of traffic analysis, with different goals (traffic identification and classification, intrusion detection, etc.). In the Cooperative Association for Internet Data Analysis web page (http://www.caida.org/home) the interested reader can find many publications about this topic, though many more exist that are not listed there.

However, we do not know of any other approach, besides our own described in [1], enabling network auditors to identify the applications that are responsible for generating traffic on a controlled network. Usually, this task is performed by an HIDS, which is co-located in the same machine of the traffic generator. However, an HIDS cannot have a wide, network-level view to detect traffic anomalies in the traffic generated by local applications (e.g. to compare with what happens in other hosts), or may be deceived or disabled by local malware.

Our work is also different from the many existing NIDS approaches (e.g. Snort [7], [8]) because these usually do not assume any strong collaboration between the hosts being surveyed and the network auditors. However, the lack of collaboration is not an advantage, but rather a drawback, as it limits the information that a NIDS can use to detect focused, host-specif anomalies from a broader, network view.

## VIII. CONCLUSION AND FUTURE WORK

In this paper we presented and architecture for managing the association between AppId's and more detailed application information, such as the path name, the binary modification time and digest and the package name and version. The whole system works without central coordination other than a central key distribution center (the PM) that deals with the identification and authentication of the involved entities.

The information provided by the traffic source hosts, through their SAR, to network auditors, such as our AR, can help the latter to detect anomalies in the former. Therefore, source hosts have all the interest to collaborate with network auditors, being this a core assumption of our work.

The detailed application information gathered by network auditors can help them a lot in the detection of abnormal traffic of one particular application (for instance, using detailed per-application traffic profiles) or to detect applications that are responsible for particular types of traffic, a capability that usually is not available to systems such as NIDS.

For future work we have still many open issues. Besides the ones pointed out throughout the paper, we still have to solve communication problems that a network auditor may face when trying to reach source hosts behind NAT boxes, such as systems running on virtual machines on top of real machines.

## ACKNOWLEDGEMENTS

This work was partially funded by FEDER through the Operational Program Competitiveness Factors - COMPETE and by National Funds through FCT - Foundation for Science and Technology in the context of projects FCOMP-01-0124-FEDER-022682 and BoDes (FCT references PEst-C/EEI/UI0127/2011 and PTDC/EEA-TEL/101880/2008).

## REFERENCES

[1] A. Zúquete, P. Correia, and H. Shamalizadeh, "Packet Tagging System for Enhanced Traffic Profiling," in *3rd IEEE Workshop on Collaborative Security Technology (CoSec 2011)*, Bangalore, India, Dec. 2011.
[2] "Secure Hash Standard," NIST FIPS PUB 180-3, Oct. 2008.
[3] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: a file system integrity checker," in *Proc. of the 2nd ACM Conf. on Computer and Communications Security*, Fairfax, Virginia, USA, 1994, pp. 18–29.
[4] J. Salowey, L. Dondeti, V. Narayanan, and M. Nakhjiri, "Specification for the Derivation of Root Keys from an Extended Master Session Key (EMSK)," RFC 5295, Aug. 2008.
[5] "Advanced Encryption Standard (AES)," FIPS PUB 197, Nov. 2006.
[6] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997.
[7] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Proc. of the 13th USENIX Conf. on System Administration*, ser. LISA '99, Seattle, Washington, 1999, pp. 229–238.
[8] B. Caswell, J. Beale, and A. Baker, *Snort IDS and IPS Toolkit*. Syngress, 2007, ISBN-10: 1597490997, ISBN-13: 978-1597490993.