Structure Matters
A new Approach for Data Flow Tracking

Enrico Lovat
Technische Universität München
Garching bei München, Germany
lovat@cs.tum.edu

Florian Kelbert
Technische Universität München
Garching bei München, Germany
kelbert@cs.tum.edu

***Abstract*—Usage control (UC) is concerned with how data may or may not be used after initial access has been granted. UC requirements are expressed in terms of *data* (e.g. a picture, a song) which exist within a system in forms of different technical representations (*containers*, e.g. files, memory locations, windows). A model combining UC enforcement with data flow tracking across containers has been proposed in the literature, but it exhibits a high false positives detection rate. In this paper we propose a refined approach for data flow tracking that mitigates this overapproximation problem by leveraging information about the inherent structure of the data being tracked. We propose a formal model and show some exemplary instantiations.**

I. Introduction

The goal of Data Flow Tracking (DFT) is to know where particular data resides within a computing system. Corresponding locations, or *containers*, for data include, among others, memory locations, files, Java objects, windows, and hardware registers. DFT has been investigated for many different *system layers* using different approaches [1]–[4]. A common idea is to mark containers with taint marks representing particular data and to propagate these taint marks according to observed system events. At any moment in time, every marked container possibly contains the data represented by its taint marks.

Taint analysis is usually performed in three phases: (1) Initial *classification* of containers; (2) *Propagation* of taint marks according to the general rule *"the result of an operation is marked with all taint marks of its operands"*; (3) *Declassification* of containers, e.g. if all content of a container is deleted.[1] The results of the taint analysis can then be used for security purposes such as data leakage prevention or enforcement of data integrity and data usage policies [5]–[7].

Because of these security goals, taint analyses tends to be conservative, resulting in many false positives. Over time, these overapproximations cumulate and lead to the so-called *label creep* situation, where all taint marks are associated with many system containers, making further data flow tracking pointless. In particular, if the usage of tracked data is constrained by policies, the system's stability might get compromised because every marked container, actually containing the data or not, would be subject to these constraints.

[1] Because a taint mark represents a policy that imposes restrictions on the usage of data, *classification* corresponds to assigning a taint mark to a container and *declassification* corresponds to removing it.

Example Scenario: Email application

In an email application, where each different part of a mail (Recipients, Subject, Body, Attachments, etc.) is modeled as a container, events causing data flows are user actions such as *send*, *reply*, *forward*, *save*, *load*, and *print* [8]. Assume a data usage policy for the message body stating "this content must not be printed". Therefore, container 'Body' will be marked with the "no-print" taint mark d3—in addition to other existing restrictions, represented by taint marks d1 and d2 in Fig. 1.

If containers with different taint marks are combined into one single container, existing DFT approaches will lead to the result that these taint marks can no longer be separated. In other words, all further operations on this single container will propagate all taint marks, even if only part of it is accessed. If the mail is saved into a file and then reopened (Fig. 1, top), a naive application of the basic DFT principle leads to a situation in which also the attachment could not be printed, because it gets erroneously marked with d3 due to overapproximation.

Instead, what we would like to achieve is presented in the bottom part of Fig. 1: After reopening a saved mail, the association between containers and taint marks should correspond to their association before the mail was saved.

Our solution works as follows: Once the mail's containers are aggregated into the single file container upon *save*, the file is marked with a special taint mark d_{new} that captures which source container is related to which taint marks (Fig. 1, bottom center). From then on, d_{new} is propagated like any other taint mark by the operating system's and other applications' DFT, e.g. upon copying, compression, or encryption. Even if these layers are not able to interpret the structure associated with d_{new}, they will preserve and propagate it. Once the file, or any copy of it, is reopened by the mail client (*load*), the structure of d_{new} is used to properly declassify each part of the mail, maintaining only the original taint marks (Fig. 1, bottom right).

Proposed Solution

Generalizing from the example, a common source of overapproximations is the *bottle-neck pattern* depicted in Fig. 2: If the content of multiple containers with different taint marks is merged into one single container, then the resulting container (*intermediate container*, from now on) is marked with all taint marks of its sources. By applying the basic rule of taint propagation explained above, further operations on this intermediate container will unconditionally propagate all taint marks, resulting in the aforementioned overapproximation.

DOI 10.1109/SPW.2014.15

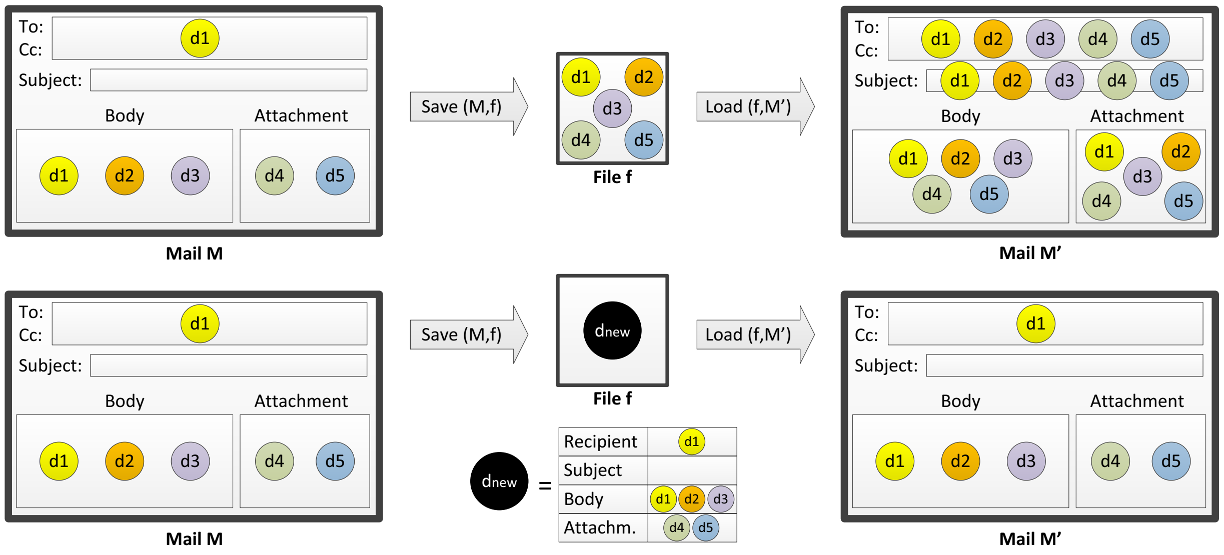

Fig. 1. Result of tracking the flow of data while saving a mail to a file and reopening it with (above) and without (below) the use of structured taint marks.

Other scenarios in which the bottleneck pattern can be observed include: compression of several files into one archive and subsequent decompression, copy-pasting data via the clipboard, and transferring content via pipes or sockets. In all of them, each destination container (like a file extracted from an archive) should only be tainted with the taint marks of one specific source (the same file before compression), rather than with all taint marks of the intermediate container (archive file).

Our solution builds on one central observation common to all of these scenarios: there exists a pair of dual *merge* and *split* operations. While a *merge* operation (such as *save*) aggregates content from different sources into one intermediate container, like c in Fig. 2, the corresponding *split* operation (such as *load*) reads the same content and separates it into multiple containers, each matching exactly one of the source containers.

The solution we propose is a generic model for data flow tracking that, in correspondence of a merge operation, marks the intermediate container with a special *structured taint mark* that represents all taint marks of the source containers. Such taint mark is then leveraged by split operations to propagate only selected taint marks to each destination, thus effectively *declassifying* the destination containers. This mitigates the label creep issue by decreasing the amount of false positives.

Note that a trivial solution for the email example is to embody the taint mark as additional content in the file while saving, and to use this information at loading time. Because we believe that DFT analysis should not interfere with the original behavior of the system, our solution achieves the same result in a *transparent* way, *never* changing the actual content. The reasons behind this choice are that (a) adding to the content may compromise its integrity or make it unusable (e.g. if a file is signed, adding content would invalidate the signature), and (b) this way our model is generic enough to be applicable at any layer and does not depend on the technical representation of a container nor on how taint marks can be embodied.

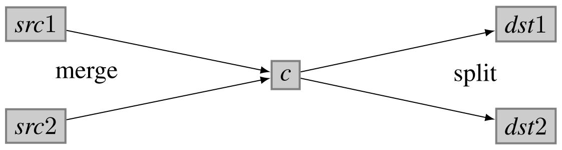

Fig. 2. Bottle-neck pattern.

Also note, that the precision of our analysis for the intermediate container is equivalent to that of basic taint propagation (cf. §III-C); the reduction in terms of false positives is in the destination containers of the split operation.

Before performing *declassification*, however, we must make sure that the structured taint mark associated with an intermediate container is *valid*: the structured taint mark must not have been propagated due to overapproximations, and the integrity of the intermediate container's content must be assured; if this is not the case, we fall back to basic taint propagation. To this end we perform additional integrity checks.

Problem. We tackle the problem of label creep in taint-based data flow tracking analyses.

Solution. We propose a generic model for taint-based data-flow tracking of structured data. Our model can be instantiated for different contexts at different system layers.

Contribution. We see our contribution in the first generic solution for event-based structured data-flow tracking. Our model *transparently* builds, propagates, and uses taint marks that reflect the inherent structure of data without semantic analysis of the tracked content. With minimal assumptions on the system and without modifying the number or the granularity of events or containers, we can increase the precision of existing DFT analyses and mitigate the label creep problem.

Assumptions. We assume the existence of dual merge and split events as explained above. These events (i) must be detectable and identifiable at runtime, (ii) must have clearly defined semantics, in particular in terms of how they propagate data, (iii) must be trusted, i.e. there must exist confidence that they behave according to the expected semantics.

Structure. We introduce the formal model underlying our work in §II. §III presents our model, formally defining structured taint marks, describing how the structure is constructed, and how it can be used for declassification. §IV describes instantiations of the model; §V compares our work to existing solutions; §VI concludes and discusses remaining challenges.

II. Formal Model

We base our model on a model for generic data flow tracking from the literature [7]. A system is described as a tuple

$$(D, E, C, F, V, \mathit{partId}, \mathit{checksum}, \Sigma, \sigma_0, \varrho)$$

where D is the set of data taint marks, E is the set of system events, C is the set of containers, and $F = F_C \cup F_P$ is the set of identifiers, where F_C is a set of container names and F_P is a set of identifiers for parts of data structures (explained in more detail in §III-A), also called partIDs.

$\underline{\mathit{partId}} : C \mapsto F_P$ assigns partIDs to containers. We use it in split operations to decide which part of a structured taint mark corresponds to a destination container. In the model partId is an oracle, while §IV describes possible instantiations of it.

$\underline{\mathit{checksum}} : (C \times D) \to V$ is another oracle that computes a checksum of the content of a specific container (e.g. the hash of a file), with V the set of all possible checksum values. The set of system states $\Sigma = (t, \mathit{struct}, \mathit{checkList})$ is defined as
(1) a <u>taint function</u> of type $t : C \to \mathbb{P}(D)$, describing which container is marked with which taint marks; if two containers are marked with the same taint mark, they are (possibly) two different representations for the same abstract data;
(2) a <u>structure function</u> of type $\mathit{struct} : D \to \mathbb{P}(F_P \times \mathbb{P}(D))$, mapping taint marks to a structure, cf. §III;
(3) a <u>checklist</u> $\mathit{checkList} \subseteq \mathbb{P}(D \times V)$, used to check whether a certain structured taint mark is valid, cf. §III.

$\sigma_0 = (\varnothing, \varnothing, \varnothing) \in \Sigma$ is the initial state. The transition relation $\varrho \subseteq \Sigma \times E \to \Sigma$ is the core of the model, as it encodes how $\sigma \in \Sigma$ must be updated in case an event occurs.

III. Structured data flow tracking

A. Structured Taint Marks

Sometimes data presents an inherent structure: a mail has a recipient, a subject, and a body; a story is divided into chapters and sections; a song into chorus and verses, etc. Although sometimes this structure is reflected by the container in which the data is stored, conceptually it remains a property of the data rather than the container. Data structure is preserved even when its concrete representation is "obfuscated", e.g. by means of compression. For this reason, our model binds the structure to the data (i.e. the taint mark) rather than to the container.

More precisely, if some data has an inherent structure, we associate its taint mark with a set of partIDs $\{\mathit{partID}_1, \ldots, \mathit{partID}_n\} \subseteq F_P$, each of which is in turn associated with a set of taint marks. The rationale is that these partIDs identify the different parts of the structured data ('Recipient', 'Subject', 'Body', 'Attachment' in Fig. 1), whilst the associated taint marks represent the data items associated with the corresponding part (e.g., d4 and d5 for 'Attachment'). Formally, the relation between each taint mark and its structure is given by the function $\mathit{struct} : D \to \mathbb{P}(F_P \times \mathbb{P}(D))$.

Associating the structure with the taint mark rather than a container has the advantage that the taint mark is then propagated independently of the type of containers in which the content is actually stored and independently of whether operations on such containers are aware of the structure or not. This allows us to easily reuse DFT event semantics described in earlier work [1], [2], [5], [7], [9], and thus to seamlessly integrate with existing DFT instantiations at different system layers. Only the semantics of those events that correspond to merge and split operations need to be updated.

Note, that our model allows us to nest structured taint marks. Also note, that if a container c is marked with a structured taint mark d, then if $\sigma.\mathit{struct}(d) = \{(p_1, \{d_1\}), (p_2, \{d_2\})\}$ the restrictions imposed by d_1 and d_2 both apply to c.

B. Merge Operations

Some scenarios allow to mitigate overapproximations by leveraging additional information about merge operations. In our terminology, merge operations are special system events that (1) aggregate data from multiple sources into a single destination container, (2) have corresponding dual *split operations*, and (3) allow us to infer information about the structure of data. The latter usually comes from external knowledge about the system, e.g. the fact that process 'zip' is an archiver. The inferred structure is associated with a new structured taint mark for the destination container (i.e. the intermediate container in a bottleneck pattern).

Formally, a merge operation $me(SRC, dst)$ merges the content of the set of source containers $SRC \subseteq C$ into the single destination container $dst \in (C \setminus SRC)$ in a structured way. This means that all taint marks associated with all source containers are grouped into multiple (possibly overlapping) sets, each of which is identified by a partID. The partIDs are derived by some properties of the set SRC, e.g. the name of the containers, and they are captured by the layer-specific function $\mathit{partId} : C \mapsto F_P$. At the model level, we use partId as an oracle to determine which parts of a structured taint mark correspond to which destination container of a split operation. The implementation of partId is instantiation-specific, cf. §IV for some examples. In contrast, the semantics of any *merge* event me can be described in a generic way:

$$\begin{array}{l}
\forall\, \sigma, \sigma' \in \Sigma, \forall\, SRC \subseteq C, \forall\, dst \in C \setminus SRC : \\
\quad (\sigma, me(SRC, dst), \sigma') \in \varrho \\
\quad \Rightarrow \sigma'.t(dst) = \sigma.t(dst) \cup \{d_{new}\} \\
\qquad \sigma'.\mathit{struct}(d_{new}) = \{(\mathit{partId}(c), \sigma.t(c)) \mid c \in SRC\} \\
\qquad \sigma'.\mathit{checkList} = \sigma.\mathit{checkList}\ \cup \\
\qquad\quad \{(d_{new}, \mathit{checksum}(dst, d_{new}))\} \\
\qquad\quad \setminus\ \{(d_{new}, v) \mid v \neq \mathit{checksum}(dst, d_{new})\}
\end{array}$$

where d_{new} represents a previously unused taint mark and $\sigma.t$, $\sigma.\mathit{struct}$ and $\sigma.\mathit{checkList}$ indicate, respectively, the taint function, the structure function, and the checklist of state σ. Performing a merge operation requires updating the list of valid checksums, which is why we replace all old checksum values of (dst, d_{new}) with the new checksum value.

While we implicitly assumed merge operations to be atomic, a merge operation might be composed of multiple subsequent events. In this case, the structure must be built incrementally; this more complex case is work in progress (§VI).

C. Split Operations

A split operation $se(src, DST)$ is the dual of a corresponding merge operation. It propagates the content of one source container $src \in C$ to a set of destination containers $DST \subseteq C$. In the bottleneck pattern, the source container src corresponds to the intermediate container. In contrast to normal taint propagation events, split operations leverage the fact that the source container is marked with a structured taint mark. As this structure was built based on information about the corresponding merge operation, split operations use this additional information to declassify the destination containers DST. In other words, split operations do not necessarily propagate all taint marks associated with the source container src to all destination containers DST, but only selected taint marks to selected containers.

For this reason, split operations do not follow the conservative approach of overapproximating data flows. Instead, they perform selected declassification of the destination containers, thus mitigating the label creep problem. Which taint marks are in fact propagated to which destination containers is determined by applying the $partId$ function to each destination container. If the result is such that a corresponding partID exists in one of the source container's structured taint marks, only the taint marks related to this partID are propagated; if no such match is found, all taint marks are blindly propagated, which is equivalent to basic taint propagation. Formally:

$$\begin{aligned}
&\forall\, \sigma, \sigma' \in \Sigma, \forall\, src \in C, \forall\, DST \subseteq C \backslash \{src\} : \\
&\quad (\sigma, se(src, DST), \sigma') \in \varrho \\
&\Rightarrow \forall\, dst \in DST, \forall\, d \in \sigma.t(src) : \\
&\quad \sigma'.t(dst) = \\
&\quad \begin{cases} \sigma.t(dst) \cup D' & \text{if } (partId(dst), D') \in \sigma.struct(d) \wedge \\ & \quad (d, checksum(src, d)) \in \sigma.checkList \\ \sigma.t(dst) \cup \{d\} & \text{otherwise} \end{cases}
\end{aligned}$$

Note, that if $(d, checksum(src, d))$ is not in $\sigma.checkList$, the integrity of the source container has been compromised and we fall back to basic taint propagation.

Let us call $closure(c)$ the set of all taint marks associated with a container c—either *directly* via tainting function t, or *indirectly*, i.e. as part of a structured taint mark which, in turn, is directly or indirectly associated with c. Note, that whenever a restriction is associated with a taint mark d, it applies to every container c that is marked with d, either directly ($d \in t(c)$) or indirectly ($d \in closure(c)$). Considering the mail example from Fig. 1, it is thus irrelevant from a precision perspective whether we taint the intermediate file container with d_{new} or with $\{d_1, \ldots, d_8\}$. The enhanced precision is only achieved at the time of the split operation.

IV. Instantiations

The model we have described is applicable to any scenario similar to the described mail example, where application-specific DFT is combined with tracking at the operating system layer. However, there are more situations in which our model can improve the precision of basic taint propagation.

Consider the action of copy-and-pasting multiple data within an application. Although the system clipboard will preserve the structure of the content, if the clipboard is modeled as a single container [5], [8], it will behave as an intermediate container in the bottleneck pattern and propagate all taint marks of the sources to all destination containers. In this case the event "copy" ("paste") corresponds to a merge (split) operation. The instantiation of $partId$ is application-specific: for a spreadsheet application it would map the cells to their 'coordinates', while for a word processor it would work on internal identifiers of sections, paragraphs, or words.

Similarly, consider a DFT analysis for the operating system [1], where containers are files, pipes and memory locations, and events are system calls. Unless there exists a ded icated monitor for the application, whenever a process reads from a file, its memory gets tainted with all taint marks of that file. These taint marks will then be propagated to every file the process writes. However, if we consider the special case of an *archive* process such as 'zip', the extraction (split) of an archive should propagate to each extracted file only the taint marks that were associated with it at the moment the archive was created (merge), rather than all taint marks associated with the archive file. Function $partId$ would map source and destination containers using their relative filenames.

Our model also applies in distributed scenarios, e.g. if many files are transferred over a TCP connection. While existing solutions [9] explain how to propagate the taint marks from one system to another, the communication channel behaves like the intermediate container in the bottle-neck pattern. In this scenario, the merge operation would be the sequence of read (from files) and write (to the socket) events observed on the "sender" side, whereas the split operation corresponds to the dual sequence on the "receiver" side. In a simple file transfer scenario, function $partId$ would map source and destination containers using filenames.

In all these examples, a simple hash of the content or a set of hash values for the different parts of the content could be used for the checksum function, to guarantee that the content to-be-split exactly matches the one at the moment of merge.

V. Related Work

Several techniques have been proposed for dynamic taint analysis, usually tailored to one specific system layer, like a programming language (Java [10], Perl [11], PHP [12]), a particular application (Internet Explorer [13], Mozilla Thunderbird [8]), a service (ESB [14]), an operating system [15], [16] or a certain hardware/machine level code. In this last category we find solutions based on binary rewriting [17]–[20], memory and pointer analysis [21], and partial- or full-system emulation [6], [22], [23]. The goal of all these solutions is a dedicated model for one particular system layer. Our model, in contrast, is generic and not bound to one specific architecture or platform, thus making it instantiable at any or all of them.

An interesting exception stems from the area of provenance aware storage systems [24], where representations of data are considered at three system layers at the same time (network, file system, workflow engine). Depending on the type of the content being handled, different tracking solutions are used. However, taint marks are propagated like in other taint analyses, i.e. without any special form of structured aggregation.

In parallel to us, Alvim et al. [25] developed an abstract model for quantitative information flow tracking that accounts for the structure of data. While promising from a theoretical perspective, no realistic instantiation of the model is presented.

Lastly, the model presented in this paper builds on top of the generic data flow tracking model presented in [7] and refines it in terms of precision. For this reason, our solution can easily be integrated in a fully-fledged usage control architecture like [2], [5], [7], [8] to support advanced policy enforcement.

VI. Challenges and Conclusions

We presented a new idea for a generic model to reduce overapproximations in taint-based data flow tracking. While an implementation is still work in progress, the core of the model is formalized and example instantiations are discussed.

In terms of limitations, this work assumes merge and split operations to be atomic. Yet, in some scenarios these operations actually correspond to sequences of events, e.g. if data is consecutively written to the same pipe. An easy solution is to model every event in the sequence using basic taint-propagation and then, in correspondence of the last event, replace the result with the structured taint mark. The drawback of this solution is that in some scenarios merge/split operations correspond to possibly infinite sequences of events, like a network stream of data. In these cases, split events may take place before the merge sequence is over. For this reason, we are working on extending our model to support an incremental building of the structure that copes with these situations.

At a technical level, additional challenges come from finding appropriate $checksum$ and $partId$ functions. In general, a basic hash function like SHA1 is enough for the checksum, but in some scenarios, like the archiver example, it is not obvious what a good choice for a $partId$ function is. Our model is general enough to support any choice, as long as $partId$ maps each source container (before merge) and its respective destination container (after split) to the same partID. Depending on the monitor capabilities and the assumptions about the archiver process, we may use just the filename or additional information, like creation date or checksum.

After extending the model to cope with the above challenges and implementing it, the next step is a proper comparison of this work in terms of performance and precision (false positives vs. false negatives) with existing DFT technologies.

Acknowledgments This work was done as part of the DFG's Priority Program SPP 1496 "Reliably Secure Software Systems," ref. numbers PR-1266/1-2.

References

[1] M. Harvan and A. Pretschner, "State-Based Usage Control Enforcement with Data Flow Tracking using System Call Interposition," in *Proc. 3rd Intl. Conference on Network and System Security*, 2009, pp. 373–380.
[2] A. Pretschner, M. Büchler, M. Harvan, C. Schaefer, and T. Walter, "Usage Control Enforcement with Data Flow Tracking for X11," in *Proc. 5th Intl. Workshop on Security and Trust Management*, 2009.
[3] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proc. 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.
[4] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical Dynamic Data Flow Tracking for Commodity Systems," in *Proc. 8th ACM Conf. on Virtual Execution Environments*, 2012.
[5] T. Wüchner and A. Pretschner, "Data Loss Prevention Based on Data-Driven Usage Control," in *Proc. 23rd IEEE ISSRE*, 2012, pp. 151–160.
[6] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *Proc. 14th ACM CCS*, 2007, pp. 116–127.
[7] A. Pretschner, E. Lovat, and M. Büchler, "Representation-Independent Data Usage Control," in *DPM*, ser. LNCS 7122, 2012, pp. 122–140.
[8] M. Lörscher, "Usage Control for a Mail Client," Master's thesis, Dep. of Informatics, Kaiserslautern University of Technology, Germany, 2012.
[9] F. Kelbert and A. Pretschner, "Data Usage Control Enforcement in Distributed Systems," in *Proc. 3rd ACM CODASPY*, 2013, pp. 71–82.
[10] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens, "Security Monitor Inlining for Multithreaded Java," in *ECOOP*, ser. LNCS 5653, 2009.
[11] "Perl Programming Documentation," http://perldoc.perl.org/perlsec.html, Accessed: 2014-02-12.
[12] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically Hardening Web Applications Using Precise Tainting," in *Security and Privacy in the Age of Ubiquitous Computing*, 2005.
[13] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic Spyware Analysis," in *Proc. of the USENIX Annual Technical Conference*, 2007.
[14] G. Gheorghe, S. Neuhaus, and B. Crispo, "xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement," in *Trust Management IV*, 2010, vol. 321, pp. 63–78.
[15] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making Information Flow Explicit in HiStar," in *Proc. 7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 263–278.
[16] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information Flow Control for Standard OS Abstractions," in *Proc. 21st ACM Symp. on Operating Systems Principles*, 2007.
[17] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting," in *Proc. 11th IEEE Symposium on Computers and Communications*, 2006, pp. 749–754.
[18] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end Containment of Internet Worm Epidemics," *ACM Trans. Comput. Syst.*, vol. 26, no. 4, Dec. 2008.
[19] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," in *Symp. on Software Testing and Analysis*, 2007.
[20] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation," in *Proc. Network and Distributed System Security Symposium*, 2011.
[21] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," *SIGPLAN Not.*, vol. 39, no. 11, pp. 85–96, Oct. 2004.
[22] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage, "Neon: System Support for Derived Data Management," in *Proc. 6th ACM VEE*, 2010.
[23] B. Demsky, "Cross-application Data Provenance and Policy Enforcement," *ACM TISSEC*, vol. 14, no. 1, Jun. 2011.
[24] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in Provenance Systems," in *USENIX Annual Technical Conference*, 2009.
[25] M. S. Alvim, A. Scedrov, and F. B. Schneider, "When not all bits are equal: worth-based information flow," Cornell Univ., Tech. Rep., 2013.