Department of
Information Engineering
and Computer Science DISI

UNIVERSITY
OF TRENTO - Italy

DISI - Via Sommarive, 14 - 38123 POVO, Trento - Italy
http://disi.unitn.it

A SURVEY OF RUNTIME POLICY ENFORCEMENT TECHNIQUES AND IMPLEMENTATIONS

Gabriela Gheorghe and Bruno Crispo

September 2011

Technical Report # DISI-11-477

Abstract

Runtime techniques bring new promises of accuracy and flexibility in enforcing security policies. While static security enforcement was previously studied and classified, this work is the first to survey the state of the art on runtime security enforcement. Our purpose is to encourage a better understanding of limitations and advantages of enforcement techniques and their implementations. We classify techniques by criteria such as abstraction level, enforced policies and security guarantees. We analyse several implementations of each technique, from the point of view of trust model, policy language and performance overhead. Finally, we discuss research issues for further investigation in policy enforcement.

1 Introduction

It is common to run applications without having their source code, and this comes with security risks. The software can be buggy or malicious, and that may not be known before executing it. For instance, on the background of Android's booming market [37], it has been shown that 28% of Android applications can access sensitive data [98], and that the risk of downloading malicious, pirated, or repackaged Android applications is very high [7]. At the other extreme, enterprise platforms (e.g., Cloud) are also subject to loss of control over running applications, since resources and data are owned by many, and it can be easy to misuse them [69]. In these cases, but also in others (e.g., copyright protection), it is vital to draw the line between allowed and disallowed application behaviour, such that malicious actions can be prevented, even if source code is not available.

Allowed and disallowed application behaviour is specified in *security policies*. Complying with security policies increases the protection of a computer system. For instance, some policies are: "accounting data must never be sent on the network","the Guest account cannot run the Disk Format utility", "inputs to this form must always be validated", or a separation of duty policy like "a bank employee cannot authorize a loan requested by him". *Enforcing* such policies means to ensure that these policies are complied with on a system where an untrusted program runs; the actions to ensure such compliance can be done before the program runs, during the program's execution, or both before and during execution. In charge of policy enforcement is a *security monitor* (also called *enforcer*, or *execution monitor*): a set of one or more mechanisms whose task is to monitor the program's execution, and react to malicious actions.

Security policy enforcement has been a flourishing domain in the area of formal methods. The theory of finite-state automata has made big steps in how to specify and enforce a large class of security policies onto an untrusted program [3, 83]. An automaton, in this respect, is a state machine that models the secure states and transitions between states for a given program; in runtime enforcement, it runs along with the program, and whenever the program is about to execute an action that is disallowed by the security policy, the automaton will raise a signal that the state transition is no longer secure. The discussion on automata for enforcement is still ongoing, with significant refinement in how an automata-based monitor can change the execution of the running program [59, 62]. However, the practical implementations of security enforcers are lagging behind the progress on the theory side: other than several isolated

examples [11, 2, 28, 44, 26], it is hard to see how much of the theoretical advances are absorbed into practical enforcement, or at least how disparate they are.

Policy enforcement seen from a static analysis perspective – checking that a program complies or not with a policy by analysing its source – has been the subject of excellent surveys in the past [78, 19, 31]. The same cannot be said for runtime enforcement – checking that a running program complies with a given policy, and taking actions if it doesn't – despite being an active research area with a broad range of applications (e.g., DRM, content validation, program monitoring, etc). This paper aims to provide security researchers, security developers and security professionals with the big picture of this domain, and an initial evaluation of the runtime enforcement tools available today.

Along with a big picture, we want to systematize the methods that exist in practice to enforce security policies at runtime. We have analysed the different approaches available to that end, and for our separation, we introduced two notions: an *enforcement technique* and an *enforcement implementation*. While a technique is a general way of solving a problem, an enforcement technique is a general way of enforcing a set of policies. Techniques that are not necessarily enforcement techniques, but are used in security implementations, are called by other researchers *security mechanisms*. An enforcement implementation instantiates an enforcement technique on a particular setting and target, and with a more particular set of tools.

From a system point of view, studying the relation between a technique, the policies it enforces, and its guarantees is a daunting task: policies can use different languages to express their constraints, the technique to enforce a policy can be implemented differently. The tradeoff is different in each case, and the researcher needs to understand the policy semantics in order to pick mechanisms that work at the policy's *locus agendi*, or scope. For instance, a policy that prevents writing to any files can be imposed on system calls (e.g., requiring system call wrapping), but can alternatively be enforced with the help of an interpreter, or a program rewriter. In each case, the scope is different (system call, runtime, or application level) hence the security guarantees over the system will differ. We are aware that in our attempt to classify different approaches, a rigorous comparison might not be possible: security policies vary greatly, just as their enforcers and their context. Nevertheless, our analysis has the advantages of covering and clustering a broad range of runtime enforcement approaches, and of providing hints of where more rigorous security evaluation methods should go, for the large amount of tools that exist to enforce security policies.

The survey proceeds as follows: after a short description of basic concepts in enforcement (Sect. 2), the paper presents the criteria we used to select the papers that we have surveyed (Sect. 3), and then the criteria used in the assessment of runtime enforcers (Sect. 4). Then, it continues with a taxonomy (Sect. 4.3) and a more detailed overview of the types of runtime enforcement techniques that were analyzed (Sect. 5, Sect. 6). The survey concludes with a high-level assessment and a discussion (Sect. 7).

2 Concepts

This section describes the main concepts used throughout the rest of the paper. First, we will refer to *the system* as the environment where the (possibly malicious) programs are running. For this survey, the system is a machine with several resources (memory, CPU, disk, etc). Within the system, some programs are trusted, some are not; still, they all request access to system resources. The untrusted programs will be analyzed by security tools to detect or correct misbehavior, and as such they will be called *targets*. We tehn define security policies as restrictions over the behaviour of targets. After a brief description of the classical concept of the reference monitor in security, which is an essential model in security inspired from operating systems, we define policy enforcement and we sketch automata as a known formalisation of the execution of a reference monitor; last, we present program analysis as an enabler of policy enforcement.

2.1 Security Policies

Bishop sees *security policy* (hereafter just 'policy') as

> "a specific statement of what is and is not allowed [..] If the system always stays in states that are allowed, and users can only perform actions that are allowed, the system is secure. If the system can enter a disallowed state, or a if user can successfully execute a disallowed action, the system is nonsecure." [14]

Another definition comes from Sterne, who draws the line between a security policy objective – a statement that protects an asset from unauthorised use – and an automated security policy, that is a particular implementable statement in line with a well-defined security objective [91]. In a more formal context, Schneider defines a security policy as a restriction over a program execution; for him, there are three main kinds of security policies: access control policies (about operations of users onto resources), information flow policies (about the data that can be inferred about a program's from observing it), and availability policies (about users denying other users the usage of a system resource) [83]. Security policies may refer to current program behaviour in relation to program history, system conditions, program context.

2.2 The Reference Monitor

The reference monitor is a conceptual model that originated in operating systems and was later adopted in security, in particular in access control. Its purpose was to prevent unauthorised users from accessing a system. Anderson's report [4] introduced the reference monitor as a concept in access control by which an abstract entity guards all accesses to a resource. Whenever a user attempts to invoke that resource, the reference monitor will intercept the invocation, determine whether it is legitimate or not, and allow it to proceed only if it verifies a set of checks. From the initial report, the reference monitor must satisfy three main properties: (1) *complete mediation* or *non-bypassability*, in that the reference monitor implementation mediates all relevant user accesses and cannot be bypassed; (2) *tamperproofness*, in that the reference monitor

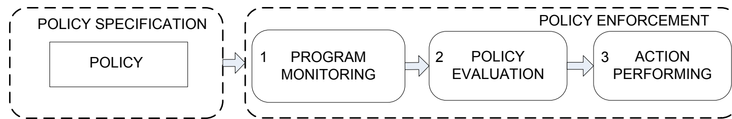

Figure 1: The basic steps for policy enforcement: once a policy is specified, security enforcement includes program monitoring, policy evaluation and action performing. The arrows mean cause-effect relations.

implementation cannot be modified (or altered) by external entities; (3) the reference monitor is small enough to allow for *verifiability*, in that the reference monitor implementation can be practically verified for correctness. The notion of reference monitor has been associated with that of *security kernel* and of a *Trusted Computing Base* (TCB). The security kernel is the concrete implementation of the reference monitor, comprising hardware, software and firmware.

2.3 Policy Enforcement

When a program misbehaves with respect to a security policy, we say that a *security (policy) violation* has occurred. *Security policy enforcement* is the set of actions, either implicitly or explicitly stated in the policy, to keep the system as compliant as possible with the policy. Since we are interested in policy enforcement that involves the program runtime, we want to investigate the existing security approaches able to check if the target adheres to the policy or not. Hence, we see security policy enforcement as a series of actions organised in three categories, or steps (see Figure 1):

1. the enforcer needs to monitor the target for policy-relevant activity (e.g., system calls, Java methods, API calls). Sometimes, monitoring can rely on *instrumentation* – modifying some part of the program to produce extra data.

2. the enforcer evaluates if the target is about to violate the security policy;

3. the decision taken in Step 2 triggers a set of actions performed by the enforcer; these actions are either punitive or remedial.

In formal methods, the most important mechanism for monitoring the execution of a target, and enforcing a policy onto its execution, is the *security automaton*. This is a finite-state automaton that models the execution by having the following elements [83]:

- a number of automaton states, out of which some are initial (at target startup);
- a number of input symbols that is set up by the security policy to be enforced;
- a transition function usually specified as transition predicates.

In Schneier's view, such automaton would execute along with the target; at each step the target would generate input for the automaton, and it would only be allowed to proceed in its execution if the automaton would be able to make a transition to a new

state based on the input, and the transition function. Schneier's model has generated a large number of formal and practical contributions in security enforcement e.g., [60, 28, 30, 44, 62, 59, 94, 2, 10, 45, 11].

Automata theory for security enforcement refers to several types of automata: the *truncation automaton*, that can recognize disallowed sequences of actions and halt the program when they are about to happen [83, 3]; and the *edit automaton*, that puts together insertion and suppression automata in order to allow for target modification when a violation is about to happen [11, 60]. Recently, the *mandatory results automata* has been proposed as a more realistic formal model of enforcement, as Ligatti and Reddy observed that the edit automata are impractical and impossible to implement in practice (edit automata assume unlimited prediction capabilities and buffer storage) [62].

2.4 The two types of program analysis for enforcement

Policy enforcement can happen before target execution, during target execution, or both before and after execution. The monitoring step in policy enforcement at execution time can employ mechanisms of a program analysis branch called *dynamic program analysis*. With hybrid enforcement that happens before and during program execution, deriving information about the program before it is run benefits from another branch of program analysis – *static program analysis*.

Static program analysis looks for patterns or properties (e.g., that the program will not generate division by zero or buffer overflows) that hold for a finite set of execution paths of a program. Static analysis considers several possible program behaviors and builds a static model of the code in order to assess properties about thecode. An analysis that cannot prove a property about the program reports it as a potential violation, and thus statis analysis is prone to false positives [19]. There are several overviews on static analysis research [78, 19]. Some enforcement techniques, e.g., information flow enforcers, use static program analysis.

Dynamic program analysis checks properties of a program based on information elicited from running the program. It uses compiled or executable code, and divides in *offline analyses* – that analyze program traces, and *runtime analyses* – that analyze the actual program execution. Unlike static analysis, dynamic analysis is less prone to produce false positives because it observes just the part of the program that executes given a particular input, and the runtime effects onto the system memory, file system, or network traffic. Analyzing dynamically only one program run at a time may be highly dependent on the input data; if the program is safe for one input set, this might not hold for another input set. The challenge for a good dynamic analysis is to test relevant inputs, and so dynamic analysis tools are prone to false negatives (i.e., the program could be declared secure when actually it is not). Some policy enforcement techniques use dynamic program analysis to reason about the target and act on it.

Dynamic and static analyses are orthogonal. An emerging trend is to use a hybrid approach, where static analysis at compile time is supplemented with employing dynamic analysis tools at runtime. There is a growing number of implementations that use both approaches (e.g., Microsoft's DebugAdvisor [8], NASA's Copilot [77]).

CRITERIA FOR ENFORCEMENT TECHNIQUES	CRITERIA FOR ENFORCEMENT IMPLEMENTATIONS
T0. OBJECTIVE **T1. ABSTRACTION LEVEL** **T2. LOCALITY** **T3. TYPE** T4. CLASS OF POLICY ENFORCED T5. GUARANTEES	I1. TRUST MODEL I2. POLICY LANGUAGE I3. OVERHEAD

Figure 2: The evaluation criteria for enforcement techniques and implementations. The criteria in bold are primary in our separation.

3 Paper Selection Criteria

This paper overviews the most important runtime enforcement techniques as well as some of their implementations. Since runtime program analysis has been active from early '90s, we have analysed over 100 academic papers published between 1993 and 2010 on various techniques and implementations in runtime enforcement. The main criteria for this selection of papers were the earliest references, the most cited papers, and their presence in well-rated conference proceedings (IEEE S&P, USENIX, CCS, SOSP, NDSS, POPL, ACSAC, ASIACCS, PLDI, OSDI, ESORICS) and journals (ACM and IEEE Transactions). A few technical reports and theses are mentioned for completeness.

We have analysed this material by splitting it into major enforcement techniques and their implementations. This separation has been done in breadth: even though an expert in a particular implementation might object to its unique placement in our classification, our effort was to capture a broad picture rather than an in-depth one.

4 Criteria for Separating Techniques and Implementations

As shown in Fig. 2, we have compared enforcement techniques from three points of view: abstraction level, objective or type of policies enforced, and locality. Then, for the implementations of each technique, we looked at the policy language it uses, its trust model, and its performance overhead. Since every implementation measures performance in its own way, comparing overheads between very different implementations is a difficult task; our aim is to give an idea over an order of magnitude rather than precise numbers.

Choosing these criteria was motivated by two viewpoints: (1) expected utility of using a technique (guarantees and type of enforcement), compared to (2) possible limitations or effort made to implement each technique. We tried to separate between features of general techniques and more specific features of some better-known individual implementations. While the former are more abstract, the latter are concrete and give a better idea over how far the implementation goes in the direction of the method.

4.1 Criteria to assess enforcement techniques

For an enforcement technique we look to evaluate the following characteristics:

T0. Objective The objective refers to what an enforcement technique is focused on protecting: either protecting the system from target behaviour, or protecting the flow of data that the target manipulates. In both cases, the untrusted program should not perform actions that are considered malicious by the policy, but the difference resides in the effect of such actions: whereas in the first case, the target corrupts the system, in the second case the target leaks data. The objective is tightly related to T4 - Class of policies enforced.

T1. Abstraction level The abstraction level refers to the *locus agendi* – scope, or location– of a technique is essential when analyzing security enforcement from a system point of view. The levels of the software stack where runtime enforcement can be located are shown in Fig. 3. There is a lower level – the operating system with system calls and memory management – and the higher level – the platform, runtime, and the application logic.

T2. Locality Locality refers to the size of the part of the program that is being analysed by a technique so that a policy is enforced [95]. For Vanoverberghe and Piessens, the locality is one event when the technique performs checks around one single event in time; when the techniques examines multiple events, then the locality is a sequence of events.

T3. Type Type refers to the effect of enforcement onto the program's execution. There are technique implementations that just flag the presence of unwanted events, and others that transform programs such that their execution is compliant with the policy.

T4. Class of policies enforced The class of enforced policies refers to what type of policy a technique can enforce. Policies have been previously split in properties (safety, renewal, liveness, soundness) and non-properties (information flow) [59]. Since the implementations we found are enforcing either access control or information flow policies, we will refer to these policies.

T5. Guarantees Guarantees refer to the assurance with which the technique enacts constraints. This assurance can refer to the guarantees given by Anderson [4], that cover complete mediation, tamperproofness and verifyability.

Most enforcement techniques use a mechanism called *interception* (or interposition) in order to detect policy relevant events (service or resource requests together and parameters). The relevant event is blocked until a decision and an action are performed, and based on the type of action, we separate between implementations whose security monitor stops executing when malicious behaviour is detected, and implementations whose monitor continues to execute. In the first case, the consequence of the stopping of the monitor can be that the monitored program is halted, or that it continues to run but throws an exception or updates a security log. We call these implementations *recognizers*. On the other hand, there are also those implementations that transform the

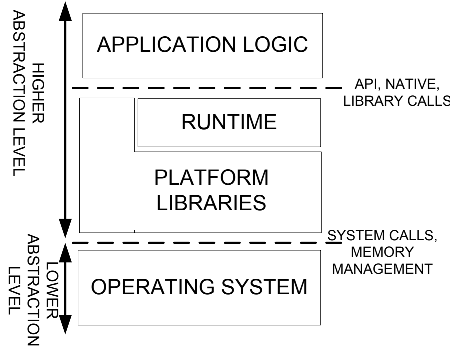

Figure 3: Abstraction levels in enforcement.

malicious target into a compliant one; we call them *sanitizers*. The security monitor, in this case, continues to execute along with the target program, but suppresses or modifies the malicious actions before they happen. It can be argued that recognizers are a subclass of sanitizers where the transformation operations are absent, but the choice to separate them is motivated by a similar distinction between traditional security automata – also known as execution recognizers – and edit automata [59] – also known as execution transformers.

Separating policies in categories, or classes, has been mentioned in a formal context. Schneider showed that not all policies are enforceable and security automata can only enforce safety properties; Hamlen showed there are several classes of enforceable properties (and policies) [46]: some that are statically enforceable, some that are runtime enforceable, some that can be enforced by runtime program rewriting, and some others that are not. Ligatti et al.[61] argued that Schneider's execution monitor, now called *recognizer*, can be more realistically extended into a monitor that can insert and suppress actions in order to correct the target program if it misbehaves. Also, while Bauer, Ligatti and Walker have classified security policies by the computational resources that are available to the execution monitor [10], Fong suggested a classification of policies from the type of information observed by the monitor, more specifically the shallow access history [34]. Fong's work is useful in practice in that it studies the impact of the limited memory onto the model of automata in enforcement.

To our knowledge, apart from these (rather few) characterisations of properties, there has been no further work in classifying types of security policies from a practical perspective. We take on the initial categories sketched by Schneider – *access control policies* and *information flow* policies, and also consider the usage control extension presented in the UCON model [76]. *Access control policies* cover the constraints on a target accessing a system resource. *Usage control policies* augment access constraints with continuous checking, attribute updates as a result of the usage of the accessed resource (i.e., state), as well as with the notions of obligations (i.e., commitments in the future) [76]. The last class of policies is *information flow policies*, that prevent data leaking to unwanted entities that could reason about program behaviour.

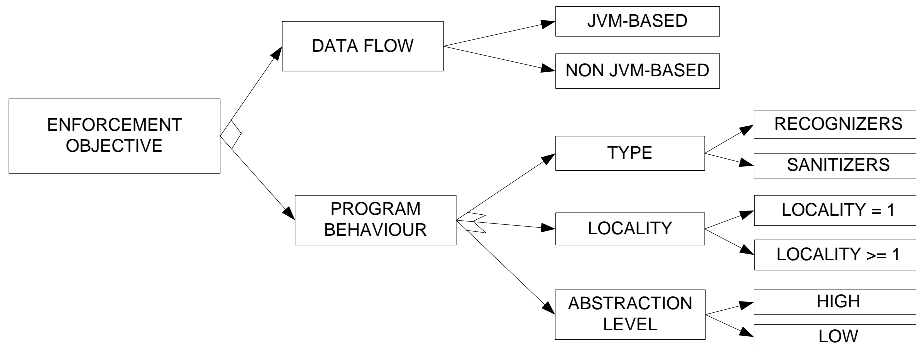

Figure 4: Two orthogonal criteria for classifying enforcement depending on policy objective: techniques enforcing policies on data flow, and techniques enforcing policies on program behaviour (with subsequent separation by type, locality and abstraction level).

4.2 Criteria to assess enforcement implementations

The features of a technique extend over the concrete implementations of the technique: abstraction level, type, locality, guarantees, objective / class of policies enforced. Still, there are some aspects of individual enforcement implementations that give a more concrete idea over the technique they implement: (I1). trust model and components; (I2). policy language; (I3). performance overhead. These aspects are tightly related to the technique: the trusted components, as well as the overhead, depend on the technique's abstraction level; the policy language is tightly connected with the type of technique, and with the class of policies that need be enforced. Figure 2 shows these aspects together.

I1. Trust model. Each implementation is defined by the system entities it trusts, what assets need to be protected and sometimes, where threats come from. To this respect, we will assess existing enforcement techniques on the quantity and manner in which they trust external components.

I2. Policy language. A security mechanism should not only be efficient but also usable by security researchers or developers. In this sense, it is important to assess the ease with which a user can write policies for a particular policy enforcement tool. This aspect refers not to the expressiveness of the policy language but to its user-friendliness.

I3. Performance overheads. Whenever runtime mechanisms are discussed, their impact on the overall application performance is important. In comparison to static enforcement, runtime enforcement technique implementations bear the risk of burdening the execution of the target every time they are used. It is generally very difficult to measure accurately the performance overhead of an implementation against another simply because the experiments use different assumptions and testbeds.

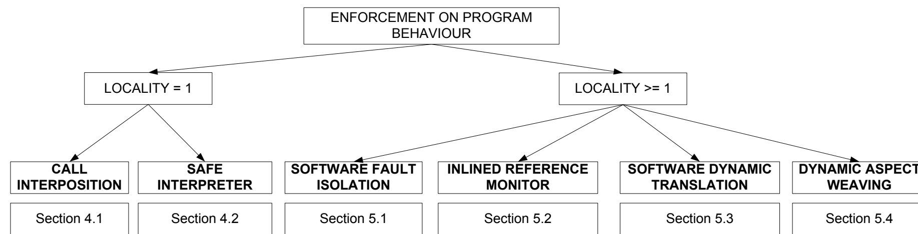

Figure 5: Taxonomy for runtime enforcement techniques on program behaviour and locality.

4.3 Taxonomy of enforcement techniques

There are several orthogonal ways by which techniques can be analysed using the criteria described in the previous section. Figure 4 and Figure 5 show several ways to evaluate enforcement techniques. Figure 4 shows that techniques can be split by their locality, their type and objective. From the point of view of the objective (or class of policy enforced), there are two types of techniques: (1) those that enforce constraints on program behaviour independent of the data that the program handles, and (2) those that enforce constraints on data handled by the program, independent of program flow. The former aim to constrain possibly malicious code from damaging the system, and are associated with *sandboxing* – a generic approach in security that aims to minimize the effects of the untrusted program over the system. These techniques can be further separated by type (either sanitizers or recognizers), by locality (either one of greater or equal to one), and by the coarse abstraction level shown in Figure 3 (high and low). The latter kind of techniques focus on data propagation. We see policies on data flow orthogonal to policies on program behaviour. Data flow techniques are usually sanitizers with locality greater or equal to one: information flow is a policy over several executions rather than an individual one and so in order to enforce information flow, a sequence of events is analysed at a time. They can further split into JVM-based approaches and non-JVM approaches.

Figure 5 looks closer at enforcers on program behaviour from the point of view of locality[1]. Locality is one for techniques that do interception and interpretation (since these execute per call or per instruction), and at least one for the other techniques. Call interposition is the enforcement technique that explicitly monitors the occurrence of certain calls – specified by the security policy – and either blocks or changes them. Safe interpreters are similar in that they execute (or interpret) one instruction or command at a time. Conversely, the techniques with higher locality are related to program rewriting: software fault isolation changes memory addresses in object or assembler code in order to prevent reads, writes, and branches to memory locations outside a policy-specified region (known as a safety domain). Software fault isolation techniques are source language independent. Inline reference monitors (IRMs) usually focus on

[1]The techniques in Figure 5 might have overlapping implementations or similar mechanisms. Our taxonomy is based on general features presented in the initial papers.

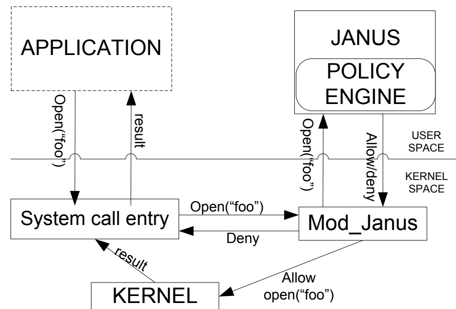

Figure 6: The Janus architecture

events closer to the application: method calls and returns, thread creation, and specific security events. IRMs work on intermediate code and hence are specific to runtimes like JVM and .NET. Software dynamic translation (SDT) for security is a more generic technique that translates from compiled to binary code at runtime, sometimes with the help of an interpreter. The added difference to interpreters is that SDT does not focus on the execution as much as on the safe translation of chunks of code into a safe version of executable code. These techniques are presented in more detail in the sections indicated in Figure 5.

5 Low locality enforcement on program behaviour

In what follows, we will provide an overview on the main techniques with unit locality and their implementations in runtime security policy enforcement. Each of them will be described and discussed in terms of its strengths and limitations, taking into account the assessment criteria enumerated in the previous section.

5.1 Call interposition techniques

Call interposition techniques exist at the system call level, but also at higher levels. They can be either sanitizers or recognizers. Hereafter we focus on system call techniques for two reasons: first, the bulk of the work in the state of the art concentrates on system call research, and second, a big part of the discussion on enforcement with system call interposition applies for calls at higher levels of abstraction.

System call interposition is motivated in that malicious behaviour can be reduced to the system calls made to access the file system, network, or other sensitive resources. Therefore, monitoring relevant system calls makes it possible to detect and counteract a broad range of malicious behaviors. Tracing system calls at runtime provides a rich amount of data: method call chains, method arguments, return values. The costs of this approach depend on where interception mechanisms reside on the host OS – either kernel-level or user-level mechanisms, and whether they block or alter the call chain.

System call interposition helps enforce access control policies and limit the domain of action for untrusted programs. System call recognizers cover both kernel-level and

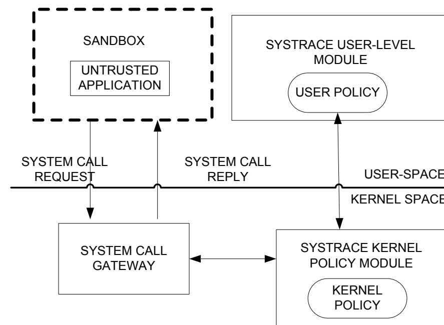

Figure 7: The Systrace architecture

user-level areas. This has a big impact on two important aspects: portability and performance. Kernel-based sandboxes can become difficult to port, since they depend on a precise kernel structure and content. Policies on intercepting system calls, either for kernel or user-space enforcers, ask for detailed knowledge on OS components, therefore are costly and difficult to develop. Enforcing such policies is also error prone when it comes to replicating OS state, races, symbolic links and related issues [35].

Higher level call interposition has the advantage that is closer to the developer's and user's understanding of the target application. It is much easier to write security constraints at a level closer to the application than to the core system. While the same system call sequence can be exhibited by two applications, a sequence of method calls is more likely to be specific to one target application. The security policy writer needs to have application-specific knowledge of the API or method calls to monitor – and this was not the case for system call level policies that are less dependent of the application that generates the calls, but more dependent on the OS used. To name two well-known examples, high-level call interceptor features are natively offered by Enterprise Java Beans interceptors in J2EE [72], as well as by Apache Tomcat [6] servlets. Some high level call interposition implementations are realized in .NET [94, 2] and in Java [40].

System call interposition is prone to tampering and bypassability problems. For instance, a monitored process cannot have more than one single monitor [33]. If a malicious user creates its own monitoring process and attaches it to any other (as a traced process), then the user would gain full control over the monitoree; this may happen because the monitor is not necessarily a trusted process – it can also be user-defined. Also, Garfinkel [35] observes that *race conditions* are the most frequent problems to occur with system call interposition. A race condition happens when parameters of a system call can change between the time they are checked by a security tool, and the moment they are retrieved by the OS to perform the system call (Time-of-Check-Time-of-Use or TCTU race). The cause is generally multi-threading or shared memory between threads, so that the shared states between operations performed by system calls can be altered. Worse, Garfinkel also shows that denying system calls may have side effects e.g., security flaws due to privilege dropping, or even undermining program reliability. These problems apply to higher-level call interposition as well.

5.1.1 System call recognizer implementations

System call recognizer implementations have been embedded in operating systems for a long time. For example, the command `strace`, the `ptrace` system call or the `/proc` virtual file system in Unix are mediator processes that can offer system call recording and filtering.

Another method of system call interception is using *call wrapping*. A wrapper can be used to trap specific calls and attach extra features to them. For instance, COLA [53] works by replacing user-space dynamic library calls with their user-customized versions, but since the aim is functionality (e.g., user notifications when system calls happen) rather than security, COLA trusts the user and is ineffective against programs that use system call instructions directly. Unlike COLA, that does not modify the kernel, approaches like SLIC [38] are more secure for system call interposition in that they are protected from malicious applications.

An example of a user-space system call filter is Janus [39] (see Fig. 6). Its focus is to confine untrusted applications by intercepting and filtering malicious system calls performed by ordinary users. Janus has a *mod_janus* kernel module that performs system call interposition, and a *janus* part that, given a policy specified by a user , decides which systems calls to allow. In case of a policy violation, Janus kills the monitored program. All processes spawned by the target are sandboxed as their parents.

A similar idea but implemented at kernel-level is presented in TRON [13]. TRON offers a discretionary access control for a single process by providing protection domains (or sandboxes) for untrusted processes: each process executes in exactly one protection domain that it cannot escape. The enforcement is done by a kernel algorithm which examines a domain's access rights to access a specific routine; if the routine is not found in a domain or if the domain does not have the needed access rights for that routine, a violation handler is invoked. The same approach is taken by ChakraVyuha [23] and SubDomain [21]: limiting the privileges of target code over system resources extends into enforcing mandatory access control for all system users.

5.1.2 System call sanitizer implementations

System call sanitizers can change the behavior of the intercepted system call by modifying the semantics of the call or its parameters. At user-level, system call modifying by wrapping is proposed by Jain and Sekar [50]. They improve the original Janus with Janus2, or J2, by reducing the number of context switches and dealing more flexibly with system calls: their solution also permits parameter modification, apart from simply system call accepting or denying.

We have found mostly hybrid system call implementations, i.e. that include both user and kernel level components. Hybrid architectures run simultaneously both kernel-level and user-level mechanisms of enforcement. They offer the combined advantages of the two approaches: the speed of the kernel-level mechanisms and the portability and ease of development of the user-level ones. Sandboxes at this level have two separate components: an interposition architecture provides access to system call data and enforces policies typically within the kernel, and a policy engine that decides on the resources the sandbox in user-space.

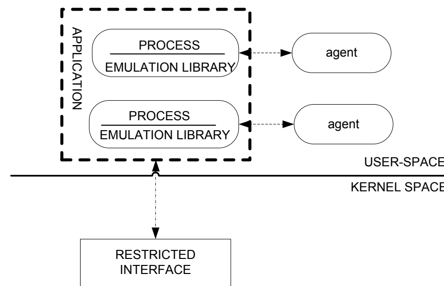

Figure 8: The Ostia architecture.

In Systrace [81], for instance, a kernel-level mechanism intercepts the system calls of a monitored process, and informs a user-level monitor about them. Systrace generates policies interactively or, more interestingly, automatically by means of a training session. The kernel policy module interposes when a system call is about to be started or finished, and retrieves information about it. If it cannot decide by itself, it asks a user-level daemon for a policy decision. This daemon monitors processes and has a more extensive view over the system. The Systrace architecture is shown in Figure 7.

Ostia [36] follows a slightly different scheme compared to Systrace and some later Janus implementations (J2 [35]). In order to invoke sensitive system resources, a typical system sends the requests of the sandboxed application to the kernel. Ostia replaces this path with a delegation of responsibility to the *sandbox broker* (see Figure 8). This broker mediates resources on behalf of the application, on the terms specified by a user policy. The TCTU race noticed before [35] is solved: while in Systrace-like architectures (approach also known as *filtering architecture*), permission checking is performed by the application sandbox and the access granting comes from the OS, in Ostia the user-level sandbox gets complete control to access resources (approach also known as *brokering architecture*).

5.1.3 Assessment over call interposition implementations

Higher level interposition implementations are not as frequently discussed in the literature as system-call interposition ones. The implications of intercepting higher level or lower level program events are discussed in [94, 92]. Erlingsson and Vanoverberghe mention that a big difference resides in the expressiveness of the policies: system-call level techniques are fine grained and their policies are expressive yet difficult to understand; application-level enforcement is coarser grained but the policies, albeit less expressive, are simpler to read and understand within the practical context of the application.

In terms of overhead, system call enforcement implementations can be fast at OS level. For instance, Janus built on SLIC's interposition mechanisms adds one extra content-switch on each potentially malicious system call, to SLIC's one-time installation cost of 2-8 microseconds plus the time for an extra procedure call. TRON incurs hundreds of microseconds that will depend the number of policies.

Criterion	System call implementations
Abstraction level	Operating system
Guarantees	mediation (full if in kernel), tamper-proof
Trusted components	OS, system call wrappers, libraries
Policy class	access control
Policy language	very low level, not user-friendly
Overheads	very low (ms), depends on no.system calls

Table 1: Summary on system call interception techniques and implementations.

In terms of security guarantees and trust model, system call interposition implementations offer tamperproofness and complete mediation if they are kernel based but not if they are user-level sandboxes. Wrapping in the second case is circumventable – a piece of code could anytime bypass a trapped call by invoking a machine-level instruction – and therefore it must be seconded by some mechanisms that prohibit subversion. The TCB includes the OS and the call wrappers. Table 1 summarizes system call interception implementations.

5.2 Safe Interpreters

A software interpreter introduces a virtual layer mediating interactions between a running program and the CPU. From a security standpoint, the biggest advantage of this approach is that untrusted programs cannot directly reach system resources: all instructions have to pass through the interpreting mechanism which can perform security checks before translating and executing the respective instructions.

Scripting languages come with their own interpreters and are highly portable since the input code is translated in the same way on any platform or environment where the interpreter is installed. For this reason, languages like JavaScript, ActiveX and VBScript are popular browser security research topics. Scripts in these interpreted languages being frequently embedded in Web pages, accessing such a page leads to the download and execution of the script by the interpreter on the user's system. This can be dangerous, hence it is important to keep such execution contained on the client side. In 1998, the notion of *safe interpreters* was proposed, with the primary aim of containing the effects of the execution of untrusted scripts [5, 73]. By Anupam and Mayer, a safe interpreter needs to ensure two things: 1) data security in the sense of confidentiality and integrity, and 2) user data privacy [5]. To achieve that, the interpreter needs to isolate scripts that might execute unsafe commands (e.g., I/O, cd, execute, open), and consequently to enforce access control for the objects in the context of the scripts. Security policies can be, therefore, about object access control, independence of contexts of various scripts, and management of trust between them.

Interpreters enforce access control policies since they react to program behaviour rather than data flow; still, it is only recently that research has been looking at information flow policies within the browser's interpreter. The level of abstraction is high since scripts and interpreted languages are programs directly written by human users, and the policies onto such program behaviour are easy to understand by a security developer.

The guarantees brought by safe interpreters are tamperproofness and mediation – but there are documented cases when the interpreter can be bypassed [5, 104].

5.2.1 Safe interpreter implementations

A well-known scripting language –Tcl– was extended with security features to Safe-Tcl [73]. Safe-Tcl employs a technique by which untrusted programs (called applets) are executed in spaces separated from the trusted system core. The applets cannot interact directly with the calling application (like an kernel space, in an OS analogy), and are kept in a sandbox of the *safe interpreter*. Safe-Tcl has two interpreters: an untrusted one called 'safe', and a trusted one called 'master'. In order to use resources, the applet needs to use 'aliases', which are commands to the master interpreter, that guards system resources and decides to allow or deny alias requests. An alias is thus a mapping between a command in the safe interpreter and a command in the master interpreter. The master interpreter has complete control not only on the states and execution of the safe interpreter, but also on how aliases are called from the applet. To this end, the Safe-Tcl security policy is a set of Tcl scripts implementing the aliases that the policy allows. The applet is completely separated from the policies, and has the power to choose at most one security policy, as an alternative to a policy-less execution when it is only allowed to perform safe commands. Safe-Tcl does not associate the 'right' policy with the 'right' applet: untrusted applications are allowed to choose between policies and this freedom is not necessary even if the policy writer is trusted: an application can choose a less restrictive policy than it should. The Safe-Tcl policies we saw (e.g, limit socket access, the number of temporary files) require blocking of unsafe actions, and hence we derive that Safe-Tcl is a recognizer implementation.

Another approach for access control enforcement with interpreters is that of SecureJS, proposed by Anupam and Mayer [5]. Unlike Safe-Tcl, where there is a master interpreter and restricted children interpreters, SecureJS focuses on the interfaces between the script and the browser, or the script and external entities like Java applets and ActiveX scripts; its focus is to restrict the external interfaces and their methods that involve scripts. It follows that this implementation's threat sources are external browser entities, unsafe scripts, while the browser and interpreter are trusted. Similar to Safe-Tcl, Secure JS concentrates on separating between namespaces with different script objects and restrictions, read-only and writable objects within a namespace, and trust shared between namespaces for object reuse (hence the TCB can include high-level objects). We see SecureJS as a recognizer: it either allows the calls between these entities to happen, aborts the call, or asks the user for permission to allow the call. Anupam and Mayer mention that policies are formulated as access control lists, but SecureJS does not come with performance overhead evaluation since the target is to repel a set of browser attacks.

Other JavaScript enforcement implementations for browser security are CoreScript [104], ConScript [65] and that by Devriese and Piessens [26]. While the first two are built to enforce access control policies, [26] looks at non-interference. CoreScript inserts security checks and warnings in the Java Script code that the interpreter will reliably execute. CoreScript's model is that of a sanitizer since it can also change the semantic of the original code, and expresses its policies as edit automata. Its TCB includes a

Criterion	Safe interpreter implementations
Abstraction level	Application level
Guarantees	tamper-proof, but not non-bypassability
Trusted components	browser, helper modules,interpreter
Policy class	access control
Policy language	scripting languages, or customized
Overheads	from 1 to 25% (peaks at 200%)

Table 2: Summary on safe interpreter techniques and implementations.

rewriting module, a policy module, and a special callback module that allows for further security validation and rewriting on runtime-generated scripts. ConScript focuses on policy safety in an adversarial environment by disallowing protected objects to flow to user code, and protecting the integrity of access paths when invoking a function; its purpose is to automatically produce expressive policies to protect a hosting Web page from malicious third-party code and libraries. It is a recognizer implementation since, from the 17 policy examples given, the action is to restrict or limit script functionality so that just allowed behaviour happens, rather than to correct possibly malicious actions.

5.2.2 Assessment over safe interpreters

An assessment over interpreters is shown in Table 2. These interpreter implementations bring mediation and tamperproofness guarantees, but non-bypassability does not always hold. As far as the security policy language is concerned, interpreters like Safe-Tcl require minute system knowledge ranging from system call API and kernel structures, to proprietary low-level APIs. The policies are expressing events at the same (high) level of abstraction with that of the mechanism enforcing the policy: the language refers to individual calls. The TCB includes the interpreter and various other callback/rewriting modules in the implementation designed to improve on non-bypassability, or to counteract certain attacks.

Safe interpreters perform worse than system call interception because of the interpretation overhead. Interpretation has a much lower performance than system call interception e.g., it raises overhead from 2 to 10% more than for compiled code. This is the main reason why interpreters are not used in complex applications. Nevertheless, the performance overhead of instrumenting interpreters for browser security seems to be much smaller – ConScript's overhead is around 1%, apart from an initialization overhead of tens of microseconds [65]. For the multi-execution technique in [26], the combination of interpreter and either serial or parallel multi-execution give execution overheads varying between 25 and 200%.

6 Higher locality enforcement on program behaviour

Higher locality for enforcement comes with **program rewriting**. Program rewriting is a technique that aims to suppress security policy violations by changing parts of the

entire program at once (rather than one instruction at a time). An extensive study over the policies enforceable by program rewriting is given in [46]. In particular, binary rewriting is interesting in our work since it happens after compilation. Binary rewriting takes an application's binaries and transforms them according to the purpose of this process, be it code optimization, code migration from one architecture to another, or binary instrumentation [103]. In this way, the enforcer is embedded in the target program. There are two types of code rewriting [105]: *static rewriting* and *dynamic rewriting*. The former changes the binaries on disk. In this case, the instrumentation is done only once for several program runs, hence the rewriting overhead is fixed. The other form of rewriting is *dynamic rewriting*, which changes the binaries in memory. The instrumentation is done at runtime, and instruction insertion or removal is done on the fly. Dynamic rewriting is sensitive to events that can only be observed at runtime as library loads, external application executions, etc. The executable is not modified, but the execution suffers from instrumentation overheads. Examples of dynamic rewriting tools are Dyninst [16], Detours [49].

Types of rewriters There are several approaches to rewriting: to contain software faults, to modify system calls or higher level calls, to control program flow, etc. We classified the following approaches: software fault isolation, inlined reference monitors, aspect weaving and control flow enforcement in the context of software dynamic translation.

6.1 Software fault isolation

Software fault isolation (SFI) was introduced by Wahbe et al. [97] and offers low-level code safety. It is a type of software address sandboxing: addresses are changed so that they fall in a specific memory region. When an application is allowed to dynamically load untrusted components, or modules, it is important that these modules do not corrupt the host system, hence the need to isolate such modules within *fault modules*. The SFI model is an encapsulation by which untrusted object code cannot operate on memory addresses different than its own range. All non-CPU system resources are accessed by system calls, so providing wrappers for the system calls to which the untrusted code can be redirected should be enough to contain malicious effects. Small observes in [89] that SFI techniques can be implemented in several ways: in a compiler pass [88], a filter between the compiler and the assembler [89], or as a binary editing tool like [97]. Managing such software-enforced spaces does not require maintaining separate address spaces, because they target a single Unix process. SFI acts at load time rather than runtime. Schneider argues that SFI is not part of the execution monitoring mechanisms because, unlike them, it modifies the target before it is actually run. Nevertheless, the instructions that SFI inserts can implement a security automaton. SFI techniques are focused at enforcing low-level access control of untrusted code to the underlying system.

From a security standpoint, SFI techniques can enforce very fine grained memory access. They also help in what is known as *control flow enforcement*. This notion relates to a known problem when executing third-party code: illegal code transfers. Illegal code transfers refer to situations when, instead of giving control to legitimate code, the system yields control to malicious code. This behavior is observed with e.g.,

return-to-libc attacks[2]; to counteract it, a security mechanism has to put restrictions on where the program counter points to before the next instruction is executed.

6.1.1 SFI implementations

The original SFI implementation of Wahbe et al. [97] offers a user-level approach to ensuring code safety and memory protection. First, it loads the untrusted code in a designated memory space. What it monitors is any low-level instruction that jumps or stores an address outside its memory area, for example procedure returns (represented as jumps to registers). A part of the target application's address space is logically separated from the rest of the space and called 'fault-domain'. Software modules placed in different fault domains can only communicate via explicit Remote Procedure Call (RPC) calls. In order not to create any resource conflicts, the OS is modified in order to know of the existence of fault domains and moreover, hardware page tables are also programmed to securely leverage data sharing between fault domains.

There are several other SFI implementations among which we noted Naccio [32], MiSFIT [89], Omniware [64]. Naccio comes in two flavours for the platform library: Naccio for Win32 and Naccio for the JavaVM. Naccio/Win32 automates the process of wrapper writing and modifies the target program too. Its abstraction level is still system calls, but the focus is on wrapping the entire platform interface. Given some resource (resources here include files, threads, network connections) descriptions and some constraints on resource usage, a policy generator automatically creates an abstract policy suited for a particular platform and purpose, together with a modified version of a platform library. Hooks are added to the target application to call the policy-enforcing library and the desired policy file, instead of the original system call entries. Omniware [64] targets mobile code safety; it is a portable virtual machine whose compiler generates portable code for an abstract virtual machine, and then translates it into native fault-isolated code at runtime. In its turn, MiSFIT [89] addresses loads, jumps, and stores, but cannot ensure protection against illegal reads. MiSFIT stands between compiler and assembler; it fault-isolates loads, stores and calls and it processes unsafe instructions by rejecting the program module, or transforming unsafe operations in safe ones. MiSFIT is hence both a recognizer and sanitizer.

For control flow enforcement, DynamoRIO focuses on dynamic code optimization and adds security by proposing the technique of *program shepherding* [52]. Program shepherding restricts execution privileges for untrusted code and also restricts program control flow. It monitors control flow transfers at runtime, by either (1) code origin, (2) instruction class, source or target. Similarly, the RIO dynamic optimizer starts off from an interpreter, and aims to achieve uncircumventable sandboxing by blocking invalid control transfers. Program shepherding is reported to prevent violating code to execute, and hence we classify it as a recognizer implementation. Further SFI implementations for control flow enforcement are NativeClient [101], and *control flow integrity*(CFI) [1]. CFI provides fine-grained integrity of the program control flow, by a mixture of static verification (that computes a flow graph), binary rewriting (that inserts runtime checks in the machine-code) and some runtime checks. CFI couples

[2]This is a buffer overflow attack that replaces the return address on the stack with the address of another instruction, usually a call in the `libc` library.

Criteria	SFI implementations
Abstraction	OS and platform
Guarantees	nonbypassability, tamproofness
Trust components	OS, rewriter, compiler
Policy class	access control
Policy language	high-level (Naccio) otherwise very low level
Overheads	low to medium, e.g., 9-45%

Table 3: Summary on some SFI techniques and implementations

well with IRMs: CFI ensures that runtime execution follows a given control graph, and this is useful for IRM to make sure the extra checks they insert in the program will be surely executed, and the security state updated. Also, CFI ensures safe regions in memory where the state keeping of the IRM can be done. While NativeClient processes x86 code to guarantees that the control flow will target just certain address ranges, CFI can restrict the flow to any address within the control flow graph. In this way, CFI efficiently blocks attacks to control-flow transfer, but cannot protect against attacks complying with the legal control flow graph (e.g., changing arguments of system calls). To solve this problem, XFI [29] builds on CFI and relies on IRM load-time verification to ensure memory access control; hence it guarantees environment integrity. We see these technique implementations as recognizers because of their blocking behaviour towards malicious code.

6.1.2 Assessment over SFI implementations

Naccio cannot enforce constrained memory accesses and code structure constraints. Its focus is on low-level safety rather than monitoring resource operations, but memory and processor usage constraints cannot be enforced. Naccio is a recognizer since it does not influence the actual behaviour of the program.

SFI implementations ensure integrity of the rewritten code, but can be circumventable: because they do not impose conditions on the flow of the program, the target can find ways of avoiding the checks of the rewriter [1]. The trusted computing base for SFI rewriters usually includes the rewriter mechanism, the compiler and the OS. Table 3 summarizes some features of SFI implementations.

As far as the security policy language is concerned, most of the implementations of SFI requires detailed and very technical knowledge of the inner system. The language refers to singular calls like loads and stores (e.g., Naccio) or groups of calls to monitor (e.g., patterns similar to buffer overflow or return-to-libc). Naccio has the problem that if the policy changes, the policy generator needs to be run again to produce an adapted version of the policy-enforcing platform library. The specification language is easy to use, because it is aimed at being accessible and platform independent. Also, Naccio's and Ariel's enforcers are triggered by single specific calls [75], loads and stores. The policies required for SFI instrumentation are very low-level.

The overheads of these implementations are not very big. This is because SFI creates logical fault domains in the context of one address space, and the remote procedure

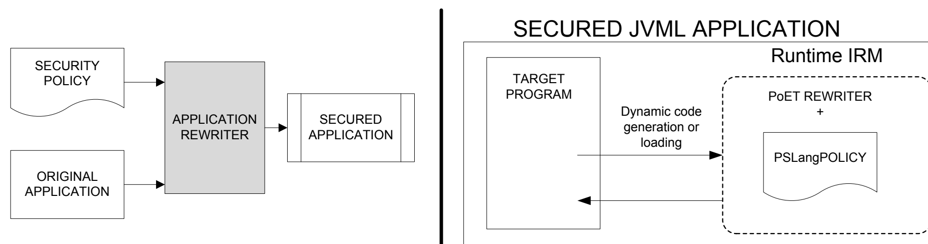

Figure 9: The inlining process on the left-hand side. The rewriter takes the application code and a security policy as input, and produces a secured version of the application. On the right-hand side, an implementation of this process with the Policy Enforcement Toolkit IRM.

calls (RPC) among these domains are fast. Omniware reports that sandboxed code runs just 9% slower than its non-sandboxed version. MiSFIT states that its read-write-call protection takes from 1.4 to 3.2 times more to execute than unprotected code [89]. The performance analysis of Naccio for JVM shows to perform better than JDK's security manager on short policies (i.e., with 84% rather than JDK's 153%), but for complex ones (e.g. limiting the number of bytes to be written or read from a location) the overheads grow dramatically due to managing resource objects in memory. Also, on SPEC2000 benchmarks, CFI is reported to have a 15% overhead compared to NativeClient's 5% overhead [101], while XFI, adding data sandboxing to CFI, should in theory be slower. Also, CFI is shown to perform faster than Program Shepherding, and on the SPEC2000 benchmarks the overhead can reach 45% [1].

6.2 The Inlined Reference Monitor

The Inlined Reference Monitors (IRMs) combine, or *inline*, execution monitoring (defined above) with the untrusted program (Fig. 9). They can act either as sanitizers or recognizers. An IRM makes use of a trusted rewriter tool that inserts security code in the target application in order to prevent any access control violations [28]. For that, it needs three types of information [30]:

- the security events are operations declared sensitive by the security policy: API calls on files, system calls, socket communication, etc.
- the security state refers to any kind of context information that the policy logic requires in order to make a decision (for example, some information from the execution history).
- the security update refers to what action the program should take whenever the security event happens, as an update to a security state. The action can be anything from a policy violation signal to a set of remedial actions (e.g., disable all socket communication).

IRMs can be recognizers and sanitizers as well. The enforcement capabilities of security automata evolved from initially halting the execution of the target to operation

suppressing or injecting. Alleviating the draconian behavior of the original security automata are the edit automata [59] that were introduced later by Ligatti et al. These new automata can truncate, insert and modify, if needed, the execution of the target. Notice here that program rewriters modify programs, while edit automata can modify steps in an execution. Introducing edit automata has led the way to more powerful monitors: edit automata can 'pretend' they allow the target to execute until the monitor eventually accepts it as legal (in other words, a sequence of untrusted actions is suppressed and only if it is proved not to violate any policy, it is reinserted in the program flow). As a result, further series of extensions have been brought to Schneider's preliminary work [46, 84, 60], by linking it to computational complexity and extending it to non-safety policy enforcement. Some notable conclusions are that intuitively, if a policy is enforceable at runtime then it is enforceable by an IRM; most statically enforceable policies are enforceable at execution time; some non-safety properties can also be enforced at runtime, depending on the computational capabilities of the security automaton modeling that execution.

6.2.1 IRM implementations

One of the first implementations to insert or inline monitoring code in the target program was SASI [92]. Security Automata SFI Implementation (SASI) extends the original SFI to execution monitoring – and to enforcement of any policy that can be expressed as an automaton. SASI embeds the policy enforcer in the executable code (x86 assembly language and Java Virtual Machine Language). The idea is that any memory-access instruction is considered sensitive, so security checks are inserted right before this instruction is executed on the target machine. These checks are non-circumventable, ensure memory safety and eliminate calls outside of the program. A rewriter module is in charge with merging of the original bytecode with the security automaton of the policy. The inserted code causes the application to halt when the automaton rejects the input, hence SASI is for us a recognizer. Because SASI can describe policies at a lower level than Naccio, it can enforce policies that Naccio cannot: constrained memory accesses, code structure constraints. Conversely, Naccio can enforce policies that SASI cannot, since it modifies the behavior of the program. Naccio's focus is on low-level safety rather than monitoring resource operations, but memory and processor usage constraints cannot be enforced.

Erlingson and Schneider introduced the IRM [83, 28, 30, 84, 46] but there are important contributions on applying or extending IRMs [94, 90, 10]. Erlingsson emphasised the use of automata with strongly-typed languages like Java and .NET [30, 84, 28]. He observed that the runtime checks performed by a Java 2 JVM consider stack inspection policies [30], while this does not hold for JVMs before Java 2. In other words, more sophisticated security policies can only be run on new JVM versions; conversely, policies not related to stack inspection could not be run on latest JVMs which do just stack inspection. The solution would be to use an inlined reference monitor to perform stack inspection, so that changing security policies would not imply changing the JVM. The suggested solution implements a Java bytecode IRM called Policy Enforcement Toolkit (PoET). The security policies for PoET are specified using a Java-like policy language: Policy Specification Language (PSLang). The integrity of the overall

reference monitor is guaranteed by the Java type safety, while the implementation is two-fold: there is a security-passing style IRM and a “lazy IRM”. The former uses a variable that stores security information from the runtime stack, and its security triggers are: method calls and returns, thread creation, Java permission checking and privileged blocks. The latter does not consider method calls and returns anymore, but manipulates the Java runtime stack and adds thread creator methods and permission checking in privileged code blocks. The gains of these implementations reside in very good performance and flexibility to do enforcement on any application event, irrespective of the JVM version.

Polymer is an approach that focuses more on policy writing and policy composition [11]. One of its novelties is that policies as Java methods that suggest how to handle trigger actions, and what to do when the suggested actions are followed. The design is similar to Naccio in that there is trusted policy compiler that compiles RMs defined in Polymer into Java bytecode, as well as a bytecode rewriter that instruments the target code according to the monitors. Polymer has the notion of a policy with other policies as parameters – called *a policy combinator*– and the syntax allows for conjunction combinators, precedence combinators, and selectors to select only one of their subpolicies. Polymer's TCB includes the policy compiler, the bytecode rewriter, and custom JVM class loaders. Polymer is a sanitizer, since it can correct or modify program behaviour.

6.2.2 Assessment over IRM implementations

As mentioned in Section 6.1, IRMs can be circumventable since they do not impose conditions on the flow of the program, so the target can avoid the IRM rewriter [1]. Nevertheless, integrity is guaranteed. For SASI on x86 code, security code integrity is achieved by employing memory protection mechanisms and by forbidding the target code to reference external entities in an uncontrolled way. For SASI on JVM, Polymer, and PoET, memory protection is by default enacted through Java type-safety; security information (e.g., the security states, the calls executed as a response to security triggers, etc) cannot be compromised because JVML forbids accessing code or data that were not loaded by the classloader.

IRMs are getting large and complex, so there is a significant increase to the TCB of the system. The trusted computing base for IRMs implementations usually includes the rewriter mechanism, as well as a certain compiler and binary analysis module. Schneider suggested leaving the IRM out of the TCB and verifying it with a static type checker, which in its turn is lightweight and does not burden the TCB as much as the whole rewriter.

In terms of policy language, security automata are a straightforward formal method to specify policies but they become difficult to grasp for more complicated policies. In SASI, having an automaton describe the policy makes a hard life to a non-expert. Still, SASI's expressiveness allows for enforcement triggers in the form of any instruction chains (e.g., division by zero, stack access properties). Polymer's Java-like syntax makes it easy to understand and use, since policy concerns are modularized and such modules are reusable; similarly but maybe not as user-friendly as Polymer, Erlingsson's stack inspection approaches have PSLang policies.

Criteria	IRM implementations
Abstraction	application and platform
Guarantees	mediation, integrity, tamproofness
Trust components	rewriter, policy compiler
Policy class	access control
Policy language	security automata or Java-like
Overheads	low to medium, e.g., 0.1-30%

Table 4: Summary on some IRM techniques and implementations

The overheads in the case of PoET/PSLang, are encouraging (as good as JVM's internal stack inspection) because the focus in on specific stack inspection primitives rather than arbitrary machine code instructions. Some initial measurements show relative overheads of between 14-60% compared to the JVM-resident stack inspection (even faster in some cases); there numbers were drastically improved in an optimized implementation so that the approach in [30] can be even faster than the JVM stack inspection. SASI's x86 performs almost as well as MiSFIT, with overheads of between 0.1 to 2.6%. For unoptimized Polymer, instrumented Java core libraries incur around 4 ms per method, and a monitored call about 0.6ms overhead.

6.3 Software Dynamic Translation

Software dynamic translation (SDT) is the translation from compiled code to binary code at runtime. SDT translates each line of code in one language into machine language instructions and, if using an interpreter, executes them. As mentioned in Section 4.3, we separate interpreters (that are eventually used by SDT) from SDT because SDT focuses on the translation of more than one instruction or command at a time, while interpreters focus on the execution of a (translated or not) instruction at a time. Safe interpreters transform the execution of an instruction, while SDT transform the whole program or parts of it. *Virtual machines* are an important example of dynamic translators; they fetch instructions, do a certain translation to the instructions, then prepare the result to be executed. There is a large number of applications of software translation: binary translation (translating binaries from one instruction set to another), dynamic optimizers, debuggers, dynamic profilers.

6.3.1 SDT implementations

Some SDT implementations that aim for security are Valgrind [70], Strata [87], DynamoRIO [52], and Java [63]. Valgrind is a powerful tool for dynamic binary instrumentation made to shadow in software every register and memory value with another value that says something about it [70]. Valgrind uses dynamic binary recompilation: (1) (re) compiles the target code, (2) disassembles the code into an intermediate form, (3) instruments, or shadows, with analysis code, and (4) converts the result into machine code again. The resulting translated code is cached and rerun if needed. Java is a SDT system where the Java interpreter stands between the bytecode to be executed and

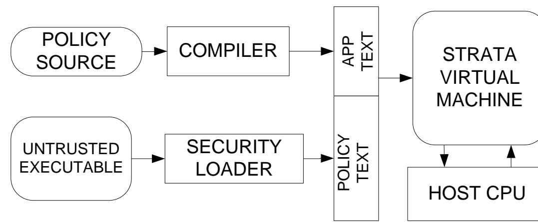

Figure 10: How policy code merges with target code in Strata.

system resources. Whether the executable code is interpreted or Just-In-Time compiled into machine code, the virtual machine is the intermediate layer that ensures portability and extra security. In Java, the interpreter only executes type-checked bytecode and these security checks are done both at runtime and compile time[3].

Another approach appears in Strata [87], which supports transparent enforcement of security policies. Strata is a virtual machine that translates executable instructions into safe operations, before executing them; user-defined policies decide whether one instruction is safe and another is not. The policy entry points are usually system calls, and an interpreter mechanism ensures that Strata executes code from within the policy. As shown in Fig. 10 [56], the policy code is merged at runtime with application code to correct unsafe behaviour, hence we see Strata as a sanitizer. In Strata, a machine code interpreter can implement a large variety of policies. It does not support a strict policy language: the claim is that any policy language can be used, provided that the developer binds it to Strata constructs. The policies in several papers have a C syntax [87, 86, 48].

6.3.2 Assessment over SDT implementations

With the Java virtual machine and binary translators, dynamic translation offers a virtualization layer that helps in both security and portability. SDT techniques in security bring strong non-bypassability guarantees e.g., program shepherding guarantees that its sandboxing around any program operation cannot be bypassed, and tamperproofness. In terms of trust model, SDT ensures that as long as the OS, policy writer and interpreter are trusted, the untrusted code cannot bypass the interpreter.

Concerning the security policy language, SDT require minute system knowledge ranging from system call API and kernel structures, to proprietary low-level APIs and security automata descriptions. SDT is approachable for security developers who can write their own call wrappers (which eventually become access control policies).

SDT implementations show much lower performance than system call interception e.g., from 2 to 10% lower. This is the main reason why software translation is not used in complex applications. For instance, Strata has been shown to bring a performance overhead of 30% which might be unacceptable. As an improvement, the concept of a Just-In-Time compiler was added to the Java implementation, and further research has been done to optimize the software translation process [85]. SDT implementations are assessed in Table 11.

[3]From this point of view, Java can be listed both as a safe interpreter and an SDT technique.

Criteria	SDT implementations
Abstraction level	runtime and application logic level
Guarantees	nonbypassable, tamperproof
Trusted comp.	OS,interpreter,compiler
Policy class	access control
Policy language	low-level, or custom
Overheads	low to medium e.g., 2-30%

Figure 11: Summary on SDT techniques and implementations

6.4 Dynamic Aspect Weavers

Aspect-Oriented Programming (AOP) is a software engineering paradigm focused on separation of concerns in software development. Some application requirements (e.g., security) are concerns that transcend the entire application, and adding, changing or removing them requires modifying the whole application. The AOP approach to solve this problem is to weave the initial application with one or more *aspects* or *concerns*. Aspects are high-level objectives and can regard customized system-wide logging, exception handling, performance optimizations, security, etc. Each aspect is associated with the specification of where and when to invoke a certain piece of code - the *advice*.

This procedure implies a number of security-related advantages [96]: a global means of performing security checking (as all the security code is located in one place), separation of the development security concerns, as the developer would not have to worry anymore about security issues along application development, and reusability of security policies across applications. An example is given in [99]: in order to insert authentication features throughout an information management system in a regular OOP way, almost all class interfaces ought to be changed, as well as implementations of existing methods that are considered sensitive.

Aspects can be woven with the application in two ways: *compile-time binding*, or static from our point of view, and *runtime binding*[4]. Static aspect weaving happens when an aspect weaver merges the source file of an aspect with the source code of a class. Once this is done, a regular compiler could compile the resulting code as normal code. An example of a static weaver is AspectJ [51]. Dynamic weaving happens at runtime and enables and disables aspects on the fly. In this case aspects can be created at different moments in time compared to the application, but their weaving together at runtime requires that the application provides some sort of weaving support. Because our work is targeted on runtime security enforcement, we will analyze runtime aspect binding.

Dynamic aspect weaving is a technique at application level. The advantages gained in this way – expressivity and flexibility in expressing and managing user-level policies – are coupled with weaker guarantees of mediation and non-bypassability. The TCB is larger since weavers have to rely on a number of lower-level mechanisms that would rewrite the code as per the desired aspects.

[4] Aspect binding can also be done at load-time, but dynamic aspect binding has received most attention.

6.4.1 Dynamic Weavers Implementations

PROSE and Wool [79, 80] are Java-based AOP frameworks for dynamic aspect weaving. The dynamic support for weaving and un-weaving aspects is located within the JVM and is called Java Virtual Machine Aspect Interface (JVMAI). This is a native interface by which the user can manage execution events related to aspect weaving: the user can register requests, get event notifications, and control what happens once a notification is raised. Whenever the currently executing instruction is a joinpoint, the execution is suspended and PROSE calls the crosscut functionality of the aspects associated with that joinpoint. At the cost of interpretation, JVMAI intercepts a wide range of fine-grained events: method calls but also field access, class loading, and breakpoint events. The lifecycle of aspects is independent of the Java application, and aspects can be (un)dynamically plugged.

There are some approaches of weaving of access control concerns, with bytecode rewriting rather than interpretation as above [45, 12, 25]. The SPoX implementation of an aspect-oriented IRM suggests a rewriting algorithm whereby first, a security class is inserted into the untrusted code, then to have all instructions that manipulate certain security-relevant objects rewritten to duplicate its state in the security class; finally, to rewrite all Java method returns to target the rewritten code that contains security checks. SPoX is a recognizer, since the detection of a security violation leads to the program termination. Alternatively, Tom, the approach presented in [25], does Java code rewriting for access control and suggests a method to map the policy environment, the access request identification, and the weaving rules, to the untrusted program. This process of mapping is an otherwise sensitive task for the Tom compiler. Tom's authorization decisions are reported to be more flexible than access allow and deny, and hence we see Tom as a sanitizer.

The aspect-based security approaches we have seen are usually formal, base themselves on security automata and mostly focus on the language capabilities of expressing security policies and how these policies can be formalized to deliver more security guarantees. Correctness is an issue: it needs to be proved that the advice, once merged into the code, enforces the specified policy and does not interfere with the property that should be satisfied by the rewritten code [24]. The verification of aspect correctness has been approached with the help of model checking in [54].

Performance enhancements to PROSE are brought by Bockisch et al., who consider another issue of aspect weaving: previous work covered crosscuts statically bound to application code (e.g., a method call). An alternative is dynamic crosscuts - those whose hooks cannot be directly associated with parts of the code (e.g.,"the control flow of variable A contains variable B", or a counter for the invocations of some methods) [15]. The problem is that dynamic aspect weaving, even as it is called dynamic, is usually logically done in a static way: 'dynamic' checks are inserted at all possible joinpoints (e.g., all instructions). This approach leaves no room for really dynamic hooks and for runtime dependencies among these hooks. Bockisch et al. suggest programmatic aspect deployment [66]to postpone advice weawving. Their architecture makes aspect weaving part of the virtual machine execution model.

Criteria	Dynamic weaving implementations
Abstraction level	application logic level
Guarantees	tamperproof
Trusted components	rewriter, interpreter, compiler, aspects
Policy class	access control
Policy language	automata, very high level
Overheads	high or very high e.g., 50-100%

Table 5: Summary on dynamic weaving techniques and implementations

6.4.2 Assessment over dynamic aspect weavers

AOP techniques present considerable overheads depending on their approach to interception and aspect weaving: reaching joinpoints, context retrieving at jointpoints, retrieving the right advice to call, calling the advice and executing it. A study by Haupt and Mezini reports some research on AOP performance based on the cost of class loading, advice lifecycle management, and various AOP benchmarks [47]. Wool incurs 4 times more time than AspectJ, and is 19% faster than dynamic code translation. The aspect-oriented IRM reported in [45] report a rewriting overhead of between 50 to 80%, but these figures depend on the policy used, the number of dynamic checks and exact weaving method.

Table 5 shows a coarse assessment over dynamic aspect weavers. Rewriters that perform aspect weaving perform very well from the point of view of policy language: the aspect language is usually easier to grasp and is dynamic, i.e., aspects can be woven or unwoven dynamically. The downsides are in performance and integrity guarantees.

The security issues of applying aspects to an application have been discussed in the state of the art [74, 99, 24]. Palmer emphasizes that in an untrusted environment, advice from unknown sources may be malicious [74] and his solution is to use *code contracts*. Sandas and Walker consider the notion of *harmless advice*, that is a block of code (an advice) required to satisfy a non-interference property; the aim is not to alter program invariants, but just use I/O capabilities and start of end certain program computations. De Win et al. bring into discussion the security guarantees of weaving aspects and those of composing security concerns [99]. Addressing this problem, the aspect-oriented inlined reference monitor [45] brings a policy specification language that merges aspects and security automata; the result is that the rewritten (or interwoven) code is proved to satisfy the desired property.

6.5 Enforcement on data flow

Policies of the type “no data from this file can be sent via e-mail” cannot be expressed by means of access control, because access control cannot track the way *any* information in a file is processed by some or any application. Information flow control (IFC) tries to bridge this gap; it imposes that unauthorized principals must not gain access to sensitive information – there must be no leakages to domains with lesser clearance. Enforcing IFC policies can be achieved with *dynamic data flow analysis*. This analysis can be performed at several abstraction levels: OS, runtime, library, and application

level. Dynamic data flow analysis associates several bits of information with a program object, follows the flow of these bits throughout the program, and if needed modifies these bits according to program statements. No entity other than the trusted monitor is allowed to change value taint value.

There are implementations using taint analysis to ensure that a target application (usually a Web application or a database) will not be subjected to attacks like SQL injections, command injections, cross-site scripting, hidden field tampering, cookie poisoning, format string attacks, or privilege escalation [18, 43, 71, 100, 22, 20, 58]. Repelling these attacks implies performing data flow control. At runtime this occurs in its three basic steps: (1) *tainting* – data from untrusted sources is marked; (2) *tracking* – all subsequent operations on tainted data are tracked; (3) *asserting* – if marked data is used in illegitimate operations, a violation is asserted. While the approach above is the same for all approaches, what differs is the abstraction level. TaintCheck [71] and Dytan [20] instrument binaries and check jump addresses and memory locations or system call arguments. Approaches like [100, 18, 58, 43] check C or Java methods.

IFC enforcement is, as of now, *mostly static*. It was only recently that dynamic taint analysis has gained ground in handling implicit flows. The state of the art of practical work in this area splits into two branches: JVM approaches and non-JVM approaches. For both types of approaches, the enforcement is realized either by rewriting or translating the application bytecode [102, 93], or by a JVM-wide interception and correction mechanism [17, 68].

6.5.1 JVM-based dynamic data flow trackers

Implementations based on the JVM aim to add information flow constraints to the Java Virtual Machine [102, 68, 42, 17, 67]. The motivation is to strengthen security controls for a language runtime that serves as a foundation for a large number of applications. Haldar et al. add object mandatory access controls and enforce them dynamically on bytecode [42]. This implies that taints are associated with Java objects, and that the security sensitive events are field reads and writes and method calls. This approach has been deemed rather coarse by subsequent works [17, 102, 68] that deal with a wider range of information flows, multithreading and exception management. The label granularity of these latest approaches is that of pieces of data (derived from files, or network resources or databases) - therefore taints can be associated with object fields, stack contexts and heap contexts [68]. One of the prominent examples in static IFC enforcement is JFlow [67], an enforcer that checks annotations to data flow both statically and dynamically; JFlow does dynamic label binding to variables, and performs dynamic access control checks for each method's access to data and its authority to declassify data values.

6.5.2 Non-JVM dynamic data flow trackers

These approaches target the compiler, runtime libraries or OS [18, 93, 55]. They usually require program source code, hence we do not see them as fully runtime approaches. For instance, some implementations perform static vulnerability analysis on the code of an application [18, 55]: a special compiler fed with an IFC policy locates

the points where the input of the program might generate policy violations. At those specific program locations, the compiler inserts calls to a runtime library and that library manages tag information as the program executes. RIFLE [93] emphasizes that, for the user, an unsafe binary is the same as a trusted one, and relies on the policy writer to protect its sensitive information. RIFLE translates the normal application binary into a binary that runs on a special instruction set architecture (a RISC ISA), which is specialized to track both implicit and explicit data flows. A completely dynamic approach is taken by TaintDroid [27], a runtime security solution for Android smartphones that does not require application source code. TaintDroid tracks the way sensitive data is handled across multiple applications, by instrumenting the Dalvik virtual machine interpreter to track data variables contained in messages sent by various applications. All taints are stored in a virtual taint map and checked at runtime when relevant events happen, e.g., a program invokes a method on sensitive data.

6.5.3 New Trends: combining IFC and IRMs

Re-evaluating Schneider's statement in an attempt to solve this problem gave rise to new *mostly-dynamic* confidentiality policy enforcement tools [57, 82]. When combining static and dynamic analysis for non-interference, Le Guernic suggests two non-interference monitors based on Edit automata and a special semantics of monitored execution. Soundness and partial transparency (the solution is unaltered by the monitor) are proven in both cases, and the novelty extends to not only detecting information flows but also *correcting* forbidden flows at runtime [41]. Sabelfeld takes the approach even further and argues that a *purely dynamic* analysis can fully enact IFC. He states that classic static analyses ignore information leaks happening if a program terminates, and proves that classic static analysis and dynamic enforcement both ensure termination-insensitive non-interference. For this property, the probability of an attacker learning a large secret (rather than one bit) is negligible in polynomial time [9].

6.5.4 Assessment of data flow tracking implementations

Most of the dynamic IFC enforcement implementations are able to deter an attacker from manipulating a faulty system by providing improper inputs. However, repelling some attacks (i.e., buffer overflows, code injections) does not necessarily imply highly secured information flows. Le Guernic observes that IFC implementations do not prove soundness and noninterference [41]. Worse, their solutions seldom discuss their degree of mediation. Approaches that modify Java bytecode [43, 102] suffer in that not all method calls are instrumented, which means that IFC applies selectively and thus the whole mechanism could be compromised by targeting native calls.

When it comes to expressing policies, usability of IFC policy languages remains a problem. TaintCheck [71] tracks tainted byte values – those from untrusted sources, or derived from tainted values – and can thus detect dangerous uses of tainted values; it can check if a particular address range or register is tainted. Such a policy is very difficult to specify by a security administrator or user without a deep understanding of the instrumented program. [18] has a similarly low-level policy language. Some JVM approaches [68, 42] follow a Java-like syntax for their policy language or a Poly-

Criterion	Data Flow Trackers
Abstraction level	OS, runtime, platform, application
Guarantees	full mediation but not always proven
Policy class	both access and usage control
Trust components	the OS, runtime or platform libraries
Policy language	complex, Java-like syntax in best case
Overheads	very high with minimum overheads around 30%

Table 6: Summary on data flow tracking techniques and implementations.

mer [12] syntax. This choice makes it easier for the Java-aware security users to write IFC policies.

In terms of performance, the overheads incurred by dynamic IFC implementations are usually important, ranging from around 30% to 200%. TaintCheck uses an x86 emulator to operate on arbitrary binaries at the x86 instruction level, and consequently that induces overheads far over 100%. While a similar performance problem applies to Dytan [20], other approaches employ static analysis on source code [18, 68] or more specific optimizations [20, 22]. The obtained performance overheads after employing static analysis range from 7% to 100% [68, 17]. Perhaps one of the best performing is TaintDroid, since it was built for a very constrained execution in terms of resources: TaintDroid reports to be only 27% slower than Android, on an IPC microbenchmark.

The TCB of most IFC frameworks includes the OS or some hardware platforms [22]. In some cases trusting the OS is replaced with trusting a modified JVM middleware, a binary rewriter, an interpreter or some emulation environment such as Valgrind [71]. RIFLE trusts the translation process and the ISA. Table 6 gives an overview of the data flow tracking implementations we surveyed.

7 Discussion on runtime enforcement

Table 7 shows the most relevant implementations we have surveyed: for each implementation we mention its type (recognizer (R) or sanitizer (S)), its operation level (high or low), and where it belongs in the taxonomy in Figure 5.

7.1 What can be observed from the state of the art

Generally, enforcers act in three main steps when enforcing a policy: one, to monitor a running program or data flowing, the second, to detect that something unwanted happened or is about to happen (e.g., program is asking for a disallowed operation, or data is about to be handled by an unwanted call), and third, to act on the target in order to produce a desired effect. For most implementations, the chosen approach is to have an alternate control program that runs along with the target, blocks relevant operations before they happen, decides their legitimacy and disallows or allows them. This enforcement model of blocking every sensitive operation is always preventive, and relies on the existence of constraints in the policy that can anticipate all symptoms

Technique	Implementation	Type	Level
Call Interposition	Cola [53]	R	low
	Janus [39]	R	low
	TRON [13]	R	low
	Ostia [36]	S	low
	Systrace [81]	S	low
	Janus2 [50]	S	low
	SLIC [38]	R	low
	ChakraVyuha [23]	R	low
	SudDomain [21]	R	low
	[94]	S	high
	[40]	S	high
Safe Interpreter	Safe-Tcl [73]	R	high
	SecureJS [5]	R	high
	CoreScript [104]	S	high
	Conscript [65]	R	high
SFI	[97]	S	low
	Naccio [32]	R	low
	MiSFIT [89]	S,R	low
	Omniware [64]	S	low
	NativeClient [101]	S	low
	CFI [1]	R	low
	XFI [29]	R	low
	[88]	S	low
IRM	SASI [92]	R	high
	[28]	R	high
	[30]	R	high
	[44]	R	high
	Polymer [11]	S	high
	[94]	S	high
SDT	Valgrind [70]	S,R	high
	Strata [87]	S	high
	Java	S,R	high
	DynamoRIO [52]	R	low
Dynamic Weavers	Prose [79]	S,R	high
	Wool [80]	S,R	high
	Tom [25]	S	high
	SPoX [45]	R	high
Data Flow Trackers	RIFLE [93]	S	low
	TaintDroid [27]	S	low
	[18]	S,R	high
	[55]	S,R	high
	[42]	R	high
	[41]	S	high
	TaintCheck [71]	R	low
	Trishul [68]	S,R	high
	Dytan [20]	S,R	high

Table 7: Summary on runtime enforcement implementations

of malicious behaviour. To counteract the limitations of such symptom prediction, the security policy could concentrate on the *effects* of a violation rather than on its symptoms. In that case, we would need an enforcement model that acts *a-posteriori*. That is, it observes that the system is in a disallowed state already, and takes action to complement that state.

Enforcement by halting the program when unwanted behaviour is about to happen is sometimes not practical, being too restrictive. Knowing what calls to intercept and process implies knowledge of the target application and also of a policy language to express policy requirements. Since different behaviours of the target might map to the same chains of calls, there are two alternatives: (1) to run the target in a secure environment where each call is monitored – but this approach is expensive – or (2) to rewrite the code of the target so that it will not misbehave. Program rewriting is supported by off-the-shelf tools (e.g., bytecode rewriters and aspect weavers), and delivers strong enforcement guarantees independent of the platform. The problem is, however, that the predominant enforcement model of rewriting is that of security automata. A security automaton is not yet fully mature to fit real-world constraints. Albeit supported by theoretical work, implementing security automata cannot yet properly *correct* the program execution when a policy is violated and cannot monitor more than one program implementation at a time. The automaton enforcement models can look at previous program executions (with some memory constraints), but when making decisions it is completely isolated – it cannot receive input from any other entity. Some other limitations of security automata are: they assume the policy is already available as an automata (which is usually difficult to obtain); their event histories may be problematic for history-based policies [28]; they correct only side-effect-free transitions and also cannot deal with information flows, concurrency and user sessions.

Policy languages tend to be similar at the same level of abstraction. A comparison among some Janus, Systrace and Ostia policies demonstrate a similar syntax in defining the sandboxed environment, with certain system paths allowed or denied, or reading or writing to the network devices. This similarity is caused by the fact that these implementations are all built on the same Unix platform, and also that their target is the same – system calls. Looking at further policy examples, the Safe-Tcl language [73] has constructs that require less deep knowledge about system internals, while Strata policies written in C from [48], seem more tedious and error prone for system call interposition. The Naccio and PSLang snippets given in [32] and [30], are perhaps more intuitive to customize for different high-level application constraints. We have not included snippets of data flow policies since they have the same semantics – i.e., to prevent classified information to leak – so the policy is 'hardcoded' into the enforcer.

Overheads grow with abstraction level and the way the mechanism works. Roughly, the further away from the machine and OS, the bigger the overheads. System call interposition implementations have the best performance (depending on the frequency of system calls); safe interpreters incur from 1 to 25%, SFI from 9 to 45%, IRMs from 0.1 to 30%, SDT from 2 to 30%, while aspect weaving and data flow trackers, over 30% overhead.

The large amount of work dedicated to securing data or information flows motivated several discussions on the capabilities and limitations of static versus dynamic techniques [17, 41]. Zdancewic argues that the challenge in information flow control

resides in motivating the need for this area of security and integrating it into existing infrastructures, with complex security policies to be enforced.

7.2 Future directions and conclusions

Policies are static. It is very difficult (sometimes impossible) to change the policies once they are set. Changing the policies to be enforced may requires changing the events to be intercepted, and monitoring mechanisms would need to be reconfigured. As a consequence, enforcement is localized – it is concentrated on one location, with a fixed sequence of actions to follow. More research is needed to make enforcement a more flexible and scalable process, with more features for security researchers or developers.

Limited type of constraints enforced. Policies enforced on the target program are limited to access control rules. There is not yet support for continuous decisions, attribute mutability and state updates. Also, concurrency is a problem for security automata, since they cannot yet enforce policies on user sessions that carry history and on concurrent programs.

Performance overheads need to be uniformly assessed. There are no common criteria by which to assess the performance penalty of similar enforcement tools, and this leaves an open door for further investigation. The performance analysis requires standardized tests (i.e. benchmarks or microbenchmarks) that pertain to the same level of abstraction, as per Figure 3. More importantly, testing the same policies written in different policy languages for different mechanisms would give the correct idea over the exact performance penalties.

The paper offered a walkthrough over the most important techniques in runtime enforcement and discussed their strengths and limitations. We feel that this work would be useful in identifying future steps in the area of security enforcement, but would also help security experts choose the right mechanisms for securing their architecture.

8 Acknowledgments

This work has been partly supported by the EU under the project EU-NoE-NESSoS.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] Irem Aktug and Katsiaryna Naliuka. ConSpec – a formal language for policy specification. *Electron. Notes Theor. Comput. Sci.*, 197(1):45–58, 2008.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21:181185, October 1985.

[4] James P. Anderson. Computer security technology planning study. *ESD-TR-73-51, Electronic Systems Division, Air Force Systems Command*, 2(1), 1972.

[5] Vinod Anupam and Alain Mayer. Security of web browser scripting languages: vulnerabilities, attacks, and remedies. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, pages 15–29, Berkeley, CA, USA, 1998. USENIX Association.

[6] Apache. Apache tomcat server. `http://tomcat.apache.org`, 2011.

[7] AppGenomeProject. App genome report february 2011. `https://www.mylookout.com/appgenome/`, 2011.

[8] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. Debugadvisor: a recommender system for debugging. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 373–382, New York, NY, USA, 2009. ACM.

[9] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002. DIKU Technical Report.

[11] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. *SIGPLAN Not.*, 40:305–314, June 2005.

[12] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *Proc. ACM SIGPLAN conf. on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM.

[13] Andrew Berman, Virgil Bourassa, and Erik Selberg. Tron: process-specific file protection for the unix operating system. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 14–14, Berkeley, CA, USA, 1995. USENIX Association.

[14] M. Bishop. What is computer security? *Security Privacy, IEEE*, 1(1):67 – 69, jan.-feb. 2003.

[15] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *Proc. 3rd intl conf. on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM.

[16] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The Intl. Journal of High Performance Computing Applications*, 14:317–329, 2000.

[17] Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. *Computer Security Applications Conference, Annual*, 0:463–475, 2007.

[18] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proc. CCS '08*, pages 39–50, New York, NY, USA, 2008. ACM.

[19] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[20] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proc. of the 2007 Intl. Symp. on Software testing and analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[21] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. Subdomain: Parsimonious server security. In *Proceedings of the 14th USENIX conference on System administration*, pages 355–368, Berkeley, CA, USA, 2000. USENIX Association.

[22] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proc. 34th annual Intl. Symp. on Computer architecture*, pages 482–493, New York, NY, USA, 2007. ACM.

[23] Asit Dan, Ajay Mohindra, Rajiv Ramaswami, and Dinkar Sitaram. Chakravyuha (CV): A sandbox operating system environment for controlled execution of alien code, 1997.

[24] Daniel S. Dantas and David Walker. Harmless advice. In *Proc. ACM SIGPLAN-SIGACT symp. Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM.

[25] Anderson Santana de Oliveira, Eric Ke Wang, Claude Kirchner, and Helene Kirchner. Weaving rewrite-based access control policies. In *Proc. ACM workshop on Formal methods in security engineering*, pages 71–80, New York, NY, USA, 2007. ACM.

[26] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[27] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[28] Ulfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004.

[29] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[30] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proc. IEEE Symp. on Security and Privacy*, Washington, DC, USA, 2000. IEEE Computer Society.

[31] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *Software, IEEE*, 19(1):42–51, Jan/Feb 2002.

[32] David Evans and Andrew Twyman. Flexible policy-directed code safety. *Security and Privacy, IEEE Symposium on*, 0:0032, 1999.

[33] Dan Farmer and Wietse Venema. *Forensic Discovery*. Addison-Wesley Professional Computing Series, December 2004.

[34] Philip W. L. Fong. Access control by tracking shallow execution history. *Security and Privacy, IEEE Symposium on*, 0:43, 2004.

[35] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, Reston, VA, USA, February 2003. The Internet Society.

[36] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *In Proc. Network and Distributed Systems Security Symp.*, pages 187–201. NDSS, feb 2004.

[37] Gartner. Gartner Says Android to Command Nearly Half of Worldwide Smartphone Operating System Market by Year-End 2012. http://www.gartner.com/it/page.jsp?id=1622614, 2011.

[38] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. Slic: an extensibility system for commodity operating systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 4–4, Berkeley, CA, USA, 1998. USENIX Association.

[39] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, Berkeley, CA, USA, 1996. USENIX Association.

[40] Tom Goovaerts, Bart De Win, and Wouter Joosen. Infrastructural support for enforcing and managing distributed application-level policies. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, volume 197, pages 31 – 43, 2008.

[41] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 218–232, Washington, DC, USA, 2007. IEEE Computer Society.

[42] V. Haldar, D. Chandra, and M. Franz. Practical, dynamic information flow for virtual machines. In *2nd Intl. Workshop on Programming Language Interference and Dependence (PLID'05)*, 2005.

[43] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Proc. 21st Annual Computer Security Applications Conf.*, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

[44] Kevin Hamlen, Greg Morrissett, and Fred B. Schneider. Certified in-lined reference monitoring on .NET. In *PLAS 06*, New York, NY, USA, 2006. ACM Press.

[45] Kevin W. Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Proc. ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 11–20, New York, NY, USA, 2008. ACM.

[46] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[47] Michael Haupt and Mira Mezini. Micro-measurements for dynamic aspect-oriented systems. In Mathias Weske and Peter Liggesmeyer, editors, *Object-Oriented and Internet-Based Technologies*, volume 3263 of *Lecture Notes in Computer Science*, pages 277–305. Springer Berlin Heidelberg, 2004.

[48] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proc. of the 2nd Intl. Conf. on Virtual execution environments*, pages 2–12, New York, NY, USA, 2006. ACM.

[49] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *In Proc. of the 3rd USENIX Windows NT Symp.*, pages 135–143. USENIX Association, 1999.

[50] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS*. Internet Society, 2000.

[51] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[52] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[53] Eduardo Krell and Balachander Krishnamurthy. COLA: Customized overlaying. In *Proceedings of the Winter USENIX Conference*, pages 3–7. USENIX, 1992.

[54] Shriram Krishnamurthi and Kathi Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16, April 2007.

[55] Lap Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[56] P. Lamanna. *Adaptive Security Policies Enforced by Software Dynamic Translation*. PhD thesis, Faculty of the School of Engineering and Applied Science University of Virginia, 2002.

[57] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automata-based confidentiality monitoring. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435 of *Lecture Notes in Computer Science*, pages 75–89. Springer Berlin / Heidelberg, 2007.

[58] Peng Li and Steve Zdancewic. Practical information-flow control in Web-based information systems. In *Proc. CSFW '05*, pages 2–15, Washington, DC, USA, 2005. IEEE Computer Society.

[59] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Intl. Journal of Information Security*, 4:2–16, 2005.

[60] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proc. European Symp. Research in Computer Security (ESORICS05*, pages 355–373. Springer, 2005.

[61] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12:19:1–19:41, January 2009.

[62] Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 87–100, Berlin, Heidelberg, 2010. Springer-Verlag.

[63] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[64] Steven Lucco, Oliver Sharp, and Robert Wahbe. Omniware: A universal substrate for web programming. In *Proceedings of the 4th International World Wied Web Conferece*, pages 359–368. O'Reilly and Associates, 1995.

[65] Leo A. Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. *Security and Privacy, IEEE Symposium on*, 0:481–496, 2010.

[66] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proc. 2nd intl. conf. on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.

[67] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[68] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, 2008.

[69] National Institute of Standards and Technology. Guidelines on Security and Privacy in Public Cloud Computing. `http://elastic.org/~fche/mirrors/www.jya.com/0003/SP-800-144.pdf`, 2011.

[70] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[71] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*. IEEE Computer Society, 2005.

[72] Oracle. Enterprise Java Beans, 3.0 specification, 2011.

[73] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The safe-tcl security model. In *Mobile Agents and Security*, pages 217–234, London, UK, 1998. Springer-Verlag.

[74] Doug Palmer. Dynamic aspect-oriented programming in an untrusted environment. In *Workshop on Foundations of Middleware Technologies (collocated with DOA'02)*. Springer Verlag, November 2002.

[75] Raju Pandey and Brant Hashii. Providing fine-grained access control for mobile programs through binary editing. Technical report, Proce. of the European Conf. on Object-Oriented Programming, 1999.

[76] Jaehong Park and Ravi Sandhu. The $UCON_{ABC}$ usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.

[77] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Proceedings of the 1st Intl. Conference on Runtime Verification*, LNCS. Springer, November 2010.

[78] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Syst. J.*, 46(2):265–288, 2007.

[79] A. Popovici, G. Alonso, and T. Gross. AOP support for mobile systems. In *OOPSLA'01 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. ACM International, 2001.

[80] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, AOSD '02, pages 141–147, New York, NY, USA, 2002. ACM.

[81] Niels Provos. Improving host security with system call policies. In *SSYM'03: Proc. of the 12th Conf. on USENIX Security Symp.*, Berkeley, CA, USA, 2003. USENIX Association.

[82] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. International Conference on Perspectives of System Informatics*, Akademgorodok, Novosibirsk, Russia, June 2009. Springer-Verlag.

[83] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[84] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101, London, UK, 2001. Springer-Verlag.

[85] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa. Overhead reduction techniques for software dynamic translation. *Parallel and Distributed Processing Symposium, International*, 11:200a, 2004.

[86] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *In CGO 03: Proc. of the Intl. Symp. on Code generation and optimization*, pages 36–47. IEEE Computer Society, 2003.

[87] Kevin Scott and Jack Davidson. Safe Virtual Execution Using Software Dynamic Translation. In *ACSAC '02: Proc. of the 18th Annual Computer Security Applications Conf.*, Washington, DC, USA, 2002. IEEE Computer Society.

[88] Scott M. Silver. Implementation and analysis of software based fault isolation. Technical report, Darmouth College, Hanover, NH, USA, 1996.

[89] C. Small and M. Seltzer. Misfit: constructing safe extensible systems. *Concurrency, IEEE*, 6(3):34–41, nov 1998.

[90] Yougang Song and Brett D. Fleisch. Utilizing binary rewriting for improving end-host security. *IEEE Trans. Parallel Distrib. Syst.*, 18(12):1687–1699, 2007.

[91] Daniel F. Sterne. On the Buzzword "Security Policy". *Security and Privacy, IEEE Symposium on*, 0:219, 1991.

[92] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA, 2000. ACM.

[93] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D.I. August. RIFLE: An architectural framework for user-centric information-flow security. In *In Proc. 37th Annual IEEE/ACM Intel. Symp. on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004.

[94] Dries Vanoverberghe and Frank Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In *Proceedings of the 10th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS '08, pages 240–258, Berlin, Heidelberg, 2008. Springer-Verlag.

[95] Dries Vanoverberghe and Frank Piessens. Security enforcement aware software development. *Information and Software Technology*, 51(7):1172 – 1185, 2009. Special Section: Software Engineering for Secure Systems - Software Engineering for Secure Systems.

[96] John Viega, J. T. Bloch, and Pravir Ch. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2), February 2001.

[97] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proc. ACM symp. on Operating systems principles*, pages 203–216, New York, NY, USA, 1993. ACM.

[98] Security Week. Study shows Android Market Outpacing Apple App Store in Growth of Apps by 3x. `http://www.securityweek.com/study-shows-android-market-outpacing-apple-app-store-growth-apps-3x`, February 2011.

[99] Bart De Win, Wouter Joosen, and Frank Piessens. Developing secure applications through aspect-oriented programming. In *Aspect-Oriented Software Development*, pages 633–650. Addison-Wesley, 2005.

[100] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proc. 15th conf. USENIX Security Symp.*, Berkeley, CA, USA, 2006. USENIX Association.

[101] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[102] Sachiko Yoshihama, Takeo Yoshizawa, Yuji Watanabe, Michiharu Kudo, and Kazuko Oyanagi. Dynamic information flow control architecture for web applications. In *Proc. ESORICS*, volume 4734 of *LNCS*, pages 267–282. Springer, 2008.

[103] Jin You, Seong Seo, Young Kim, Jun Choi, Sang Lee, and Byung Kim. Kimchi: A binary rewriting defense against format string attacks. In Jooseok Song, Taekyoung Kwon, and Moti Yung, editors, *Information Security Applications*, volume 3786 of *Lecture Notes in Computer Science*, pages 179–193. Springer Berlin - Heidelberg, 2006.

[104] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 237–249, New York, NY, USA, 2007. ACM.

[105] Jingyu Zhou and Giovanni Vigna. Detecting attacks that exploit application-logic errors through application-level auditing. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 168–178, Washington, DC, USA, 2004. IEEE Computer Society.