A Model for Automatically Repairing Execution Integrity*

Michael E. Locasto, Gabriela F. Cretu, Angelos Stavrou, Angelos D. Keromytis
Department of Computer Science, Columbia University
{*locasto,gcretu,angel,angelos*}@cs.columbia.edu

Abstract

Many users value applications that continue execution in the face of attacks. Current software protection techniques typically abort a process after an intrusion attempt (*e.g.*, a code injection attack). We explore ways in which the security property of integrity can support availability. We extend the Clark-Wilson Integrity Model to provide primitives and rules for specifying and enforcing repair mechanisms and validation of those repairs. Users or administrators can use this model to write or automatically synthesize *repair policy*. The policy can help customize an application's response to attack. We describe two prototype implementations for transparently applying these policies without modifying source code.

1 Introduction

Systems that sustain their availability in the face of attack are, in some sense, considered secure. This paper explores the use of the security property of integrity to provide availability in cases where faults and attacks can be accurately detected. Previous approaches to detecting attacks against applications typically terminate the attacked process without considering how to repair the integrity of the execution and safely continue processing. Few alternatives to crashing exist[1]. Crashing is a frustrating experience, and many users dislike having their writing or browsing session abruptly terminated, with no opportunity to save work. Even if software could somehow ignore the attack itself, the best sequence of actions leading back to a safe state is an open question.

Automating such a response strategy is challenging, as it is often unclear exactly how a system should recover from any particular attack. Furthermore, a system may require dynamic recovery; that is, the exact response may vary depending on the state of the system as well as the nature of the attack. Therefore, a strictly static response seems unsuitable. Although recent work has described methods for automatically and transparently recovering from faults and attacks [23, 26, 24], these approaches propose automatic fixes that are based on specific heuristics[2].

We articulate a way to augment these approaches with the notion of *recovery policy*. A recovery policy (or a repair policy – we use the terms interchangeably) describes how execution integrity should be maintained after an attack is detected. Even if we cannot guarantee a completely automated recovery, we can provide a way for a user to customize an application's response to an intrusion attempt. For example, we

*This research was supported by NSF grant NSF CCF-05-41093.

[1] In some environments, crashing is the best response, but we assume situation exist where a user desires continued execution.

[2] There is nothing inherently wrong with using heuristics. In fact, they may be the only valid methods for responding to faults or vulnerabilities that are present because of unclear specifications in the tools and languages used to build systems.

show how a user can specify and choose a number of recovery actions for Firefox, including saving all open tabs, in response to the exploit of a real vulnerability.

In order to provide a theoretical framework for recovery policy, we extend the Clark-Wilson Integrity Model (CW) [7] to include the concepts of (a) repair and (b) repair validation. CW is ideally suited to the problem of detecting when constraints on a system's behavior and information structures have been violated. Our extension supplements the model with primitives and rules for recovering from a policy violation and validating that the recovery was successful.

1.1 Motivation

Given the automated nature of attacks, synthesizing and executing a recovery sequence must also be automated. The very nature of automation, however, makes system owners understandably reluctant to adopt these countermeasures, as the system may generate a semantically incorrect response. Therefore, any viable response system must behave like a flexible behavior firewall for both anticipated and unanticipated errors and attacks. We are motivated by this challenge.

1.1.1 Semantically Correct Healing

At least two systems have suggested ways to execute through a fault by effectively pretending it can be handled by the program code or by some instrumentation to keep track of undefined memory accesses [26, 24]. However, these and similar strategies give rise to semantically incorrect responses.

Figure 1 illustrates a specific example: an error may exist in a routine that determines the access control rights for a client. If this fault is exploited, and the self-healing machinery glosses over the fault, it may return a value that allows the authentication check to succeed. This situation occurs precisely because the recovery mechanism is oblivious to the semantics of the code it protects. One solution to this problem relies on annotating the source code to indicate which routines should not be "healed", but we find this technique unappealing. Instead, we propose the next logical step. Since source-level annotations serve as a vestigial policy, we formalize the notion of a repair policy and create a system for specifying and enforcing it without modifying source code.

```
int login(UCRED credentials)
{
  int authenticated = check_credentials(credentials);
  if(authenticated) return login_continue();
  else return login_reject();
}
int check_credentials(UCRED credentials)
{
  strcpy(uname, credentials.username);
  cred_OK = checkpassword(lookup(uname), credentials);
  return cred_OK;
}
```

Figure 1: *Example of Semantically Incorrect Response*. If an error arising from a vulnerability in `check_credentials` occurs, a self-healing mechanism may attempt to undo the effects of `check_credentials` and return a simulated error code. Any value other than `0` that gets stored in `authenticated` will cause the login to succeed. What may have been a simple DoS vulnerability has been transformed into a valid login session by virtue of our "security" measures. This paper explores a method for intelligently constraining return values and other application data.

1.1.2 Constraining Program Behavior

The rational behind our integrity-based approach to self-healing is most easily understood by sketching its evolution from the initial idea. We began by thinking about the possibility of overloading a standard error reporting mechanism in systems programming languages: the `assert` procedure. An `assert` procedure evaluates the logical condition supplied as its argument. The semantics of this procedure typically make it useful for fault detection and notification: if the condition evaluates to false, the routine reports an error and subsequently aborts execution. Our initial approach was a thought experiment about changing the design of `assert` routines to *repair* as well as detect and notify. A simple approach would attempt to adjust the values of the relevant program state (*i.e.*, state involved in the condition supplied to the routine) to make the asserted expression true. Automatically converting existing `assert` macros (either via runtime instrumentation or altering the compiler or preprocessor) to these Self-Correcting Assertions (SelCA) is one possible way to provide a self-healing mechanism. Furthermore, it is simple to express constraints on return values via this mechanism, and potentially handle the difficulty illustrated in Figure 1.

Unfortunately, this approach suffers from a number of deficiencies. First, `assert` routines most likely exist as a last resort mechanism to forcibly crash the program if the asserted condition fails. Changing these semantics by directly overloading them will interfere with that goal. Second, invocations of `assert` macros actually evaluate the supplied expression. These semantics are problematic if the evaluated condition has side effects[3]. Using assertions for self-healing is dangerous in this situation because the very mechanism that provides security alters program state in an uncontrolled way. Another obstacle arises when legacy systems lack assertions altogether (this is the case for at least one modern programming language, Java). Finally, this form of SelCA is similar to other forms of self-healing in that it is effectively a system-level hack without any supporting formal model.

1.2 Challenges

This three-part problem motivates our work. First, since the source-level placement of most assertions is unlikely to prove useful for self-healing, more suitable instrumentation points must be identified. Implicit in this challenge is the need to specify the constraints to be evaluated at each of these points. Second, it is necessary to speculatively execute a constraint and undo any side effects if it fails. Third, the lack of a formal model makes it difficult to compare various self-healing approaches and to reason about the properties of these systems in general. We defer the second challenge to future work and note that many systems discuss techniques to rollback and replay execution (see [12, 5] for overviews). Instead, this paper focuses on the first and third challenges:

1. specifying a formal model of repair policy
2. generating the content of a policy (*i.e.*, the constraints)
3. identifying places to instrument the application (*i.e.*, insert calls to evaluate the policy).

Even though vanilla assertions are not suitable for addressing these challenges, the intuition is still sound: assertions provide an opportunity to self-heal because the code's author felt that a particular constraint was critical to continued correct operation. Asserted constraints also compare favorably to normal mechanisms

[3]The `assert` manual page notes: "`assert()` is implemented as a macro; if the expression tested has side-effects, program behaviour will be different depending on whether NDEBUG is defined. This may create Heisenbugs which go away when debugging is turned on."

for dealing with system errors, such as structured exception handling. Finally, constraints also provide a good indication of the complexity of the self-healing problem, and Section 4 presents a survey of assertion use in a variety of popular applications.

1.3 Requirements for Recovery Policy

Our overall goal is to automate the recovery process by limiting the amount of human involvement during repair. Once we understand the requirements, we can begin to design ways to automate parts of the model. A recovery model requires explicit, measurable, and enforceable constraints on the integrity of program execution. The model must contain primitives that describe basic data types as well as operations on data items of those types. The model must describe mechanisms for measuring the integrity of these data items as well as repairing that integrity by returning data and other execution artifacts to a valid state.

Recovery requires a measure of validity. The model should provide a mechanism for ensuring that the repair operation was valid. Automatic repair attempts to anticipate the combined intent of both the system designer and the user. In some cases, healing may not be possible, and the model should be able to express a range for the quality of the repair.

Recovery requires context. Existing detection and exception-handling mechanisms often lack specific information to diagnose and correct the core problem automatically. Analysis of the fault is often delegated to a manually-driven forensics process, thus slowing the response rate of the system.

Recovery requires separation of duty (SoD). A recovery model should cleanly separate the processes of policy specification and policy enforcement. Policy defines a particular arrangement of the model primitives, creates an instance of the model, and ties it to a particular implementation. The notion of separation of duty goes further than simply separating policy from mechanism. For example, an attack may have extensively altered program state before being detected. If the integrity repair system is intertwined with the protected system, the repair system itself may have been damaged or corrupted. In this sense, the term "self-healing" is somewhat of a misnomer. Separate entities should implement and enforce the mechanisms in the repair model in order to minimize conflicts of interest as well as attacks on the repair model itself.

1.4 Contributions

The main contributions of this work complement and enhance previous work in self-healing systems. First, we construct a formal model of integrity repair by extending an established integrity model: Clark-Wilson. Second, we show how this model is useful for providing a *customizable* response to intrusion attempts. We also consider how to automatically generate policies for one complexity class of constraints as well as demonstrate manual policy creation. Third, we sketch two prototypes (one for *x86*, one for Java bytecode) that can transparently insert the enforcement of these policies into real software without modifying the source code (although mapping constraints to source-level objects assists in maintaining application semantics). We expect the repairs offered by this "behavior firewall" to be temporary constraints on program behavior – emergency fixes that await a more comprehensive patch from the vendor.

Our contributions include a few that are philosophical in nature. First, we show how related systems and previous work can be classified by our model, thus demonstrating its generality. Second, we briefly consider the differences between a runtime repair system based on assertions and one based on exceptions. Systems should include structured forensic information in order to assist attempts at recovering execution rather than leave the programmar with little more than a notification and stacktrace.

2 A Model for Integrity Repair

The Clark-Wilson integrity model [7] (CW) formalizes the notion of information integrity. A policy that uses the model describes the relationships between principals, data items, and operations on those data items. Clark and Wilson developed the model to illustrate that then-current integrity models were better suited to the confidentiality requirements of multi-level security systems. Clark and Wilson use the running example of business accounting principles (*e.g.*, double-entry bookkeeping) throughout their analysis. Their model provides enough formalism to describe how these longtime manual procedures can be implemented in a computerized context. The CW model also fulfills the requirements in Section 1.3.

2.1 The Clark-Wilson Integrity Model

The basic data type in CW is a Constrained Data Item (CDI). A CDI has a logical relation constraining its value range. There are two types of procedures that can operate on a CDI. A Transformation Procedure (TP) is a well-formed transaction that transitions the system from one valid state to another. A TP processes sets of CDIs on behalf of a particular authenticated user. An Integrity Validation Procedure (IVP) is used to measure whether a CDI conforms to the integrity specification. In real software systems, TPs are analogous to functions that operate on data structures. IVPs roughly correspond to mechanisms like double-entry bookkeeping in financial systems. For software security, IVPs map to detection mechanisms like stack canaries [9, 13], taint-tracking [8], program shepherding [17], address space randomization [4, 29] and array bounds checking in languages like Java. In these cases, the CDIs in the system are data constructs like stack-resident return addresses. CW applies equally well to these low-level data items as well as more complex data structures. Of course, not all data in a system is constrained. In these cases, data (*e.g.*, from the user, adversary, or the environment), is represented by an Unconstrained Data Item (UDI). Any TP taking a UDI as input must ensure that it transforms all possible values of a UDI to a CDI. Many system vulnerabilities arise from the operation of an incorrect TP on a UDI, or data derived from a UDI.

The core of the model is a collection of nine rules that deal with enforcement (E) and certification (C) of users, IVPs, TPs, CDIs, and UDIs. For reference, we reproduce the rules in Appendix A. Some rules are annotated with shorthand names that sum up their purpose.

The heart of any specific policy expressed in CW terms is the list of the relations specified in rule **E2**. The remainder of the rules simply ensure that the model works as expected. The basis of the model relies on manual certification of TPs and IVPs. Of course, if all such procedures did not contain vulnerabilites simply by virtue of certification, the security community would have little to worry about. Certification is performed by humans on inherently complex constructs and is therefore susceptible to error. Furthermore, the mechanisms proposed by the model must themselves be protected against tampering. Rule **E4** helps to ensure this, but it still depends on a manual process. Clark and Wilson note this and suggest minimizing certification rules. They also note that many systems do not separate policy and mechanism: application-specific policy is encoded in TPs (an observation especially true for error-handling code).

2.2 Integrity Repair

The CW model focuses on detecting and preventing unauthorized data modification. Although a TP should move the system from one valid state to the next, it may fail for a number of reasons (incorrect specification, vulnerability, hardware faults, *etc.*). The purpose of an IVP is to detect and record this failure. CW does not address the task of returning the system to a valid state or formalize procedures that *restore* integrity. In contrast, we concern ourselves with ways to recover after an unauthorized modification.

The straightforward nature of the CW model does not require a great deal of added complexity to increase its power to describe the repair of failed integrity constraints. Our extensions to the model include two new types of procedures: Repair Procedures (RP) and Repair Validation Procedures (RVP). We use our additions to create a *recovery policy*. This part of the integrity policy specifies how to return to a valid state; that is, it describes what changes to the system must occur in order for a set of IVPs to measure some set of CDIs and return "VALID."

In the example of Figure 1, variables like `authenticated`, `uname`, and `auth_OK` are CDIs. The `login`, `strcpy`, and `check_credentials` functions are TPs. Since manipulation of `uname` may overwrite a stack return address, any standard detection mechanism for this type of exploit serves as an IVP for `uname`. If the IVP indicates that the TP has failed to maintain the integrity of this CDI, then a RP is needed that will set an appropriate value (presumably FALSE or reject) for the CDIs `auth_OK` and `authenticated`. Furthermore, a RVP should be invoked to test the results of this repair. The RP helps ensure that the value of the CDIs aren't simply set to a satisfying or legal value, but rather the most appropriate value given the current system state.

Integrity Repair Model Rules Our additions to the CW model are based on four new rules that supply repair and repair validation procedures, although the **C7** rule can easily be combined with rule **C3**.

1. **C6:** All RPs and RVPs must be certified valid. For RPs, certification indicates that it returns a given set of CDIs to a valid state. For RVPs, certificaion mirrors that of rule **C1**. For each RP and RVP and each set of CDI that they may manipulate, the security officer must specify a "relation" which defines that execution. A relation is of the form: $(RP_i \vee RVP_i, (CDI_a, CDI_b, CDI_c, \ldots))$, where the list of CDIs defines a particular set of arguments for which the RP or RVP has been certified.
2. **E5:** The system must maintain a list of relations of the form: (USERID, RP_i, $(CDI_a, CDI_b, CDI_c, \ldots)$ which relates a user, a RP, and the CDIs that a RP may adjust back to a satisfactory value assignment.
3. **E6:** The system must maintain a list of relations of the form: (USERID, RVP_i, $(CDI_a, CDI_b, CDI_c, \ldots)$ which relates a user, a RVP, and the CDIs that a RVP may measure to make sure they have a satisfactory value assignment.
4. **C7:** The list of relations in **E5** and **E6** must be certified to meet the separation of duty requirements. For example, an RVP must be implemented and certified by a user other than the user that certified the corresponding IVP and RP. This SoD is useful in case either of these procedure was maliciously or incorrectly certified.

```
reference_monitor(TP t, CDIList c) {
  IVP ivp = lookupIVP(t,c);
  boolean valid = ivp(c);
  if(!valid)
    RP rp = lookupRP(c);
    RVP rvp = lookupRVP(c);
    rp.execute(c);
    boolean verified = rvp(c);
    if(!verified)
      abort();
  return;
}
```

Figure 2: *IVPs, RPs, and RVPs as an Inline Reference Monitor.* The IRM can fall back to aborting execution if the repair procedure does not succeed in restoring integrity to the CDI set.

2.2.1 Model Implications

Note that our model does not constrain the relationship between RPs and TPs. Rather, there is an indirect mapping through the CDIs that form the relations of **E2** and **E5**. Consequently, we leave as an implementation choice which TPs an IVP should be invoked for, and this situation naturally reflects the choice of protection/detection mechanism in use and where (*i.e.* how often) it is executed. Selecting which TPs to monitor depends on the choice of defense instrumentation, and strategies range from random selection to full coverage (invoking an IVP after each TP). Other interesting enforcement choices include selection based on proximity of a TP to input handling routines or the error history of a TP. RVPs are also associated with a set of CDIs and serve to to check the validity of the CDIs after the application of an RP. Figure 2.2 expresses the arrangement of TP, IVP, RP, and RVP as a form of an Inline Reference Monitor. The model also frees us from having to modify the source code; the invocation of the IRM can be inserted in a program binary.

2.2.2 Satisfying Requirements

We next explore the extent to which our model satisifies the requirements from Section 1.3. The model serves our overall goal quite well: human involvement during repair is limited, as repair procedures are carried out automatically, although their synthesis may have to be non-automatic in some cases. CW provides concrete and measureable constraints that are certified to be enforceable (IVPs). CW provides separation of duty as one of its core constructs: integrity is easier to maintain if trust is not vested wholly in one principal. SoD breaks down the actions of a transaction so that prinicipals must conspire to subvert it. Simple randomized schemes provide less onerous coordination and a probabilistic rate of failure for any collusion. Our modifications satisfy the requirements for "repairing [the] integrity by returning data and other execution artifacts to a valid state" through the use of RPs. Context for the repair actions is captured via the set of CDIs that should be fixed, providing a precise snapshot of the relevant state. Finally, the RVP construct provides a mechanism to measure the validity of the repair operation.

2.3 Constraint Repair

The heart of the CW model and our extensions to it are lists of relations that constrain the values of CDIs. The constraining relation on a CDI may be arbitrarily complex. In this paper, we limit our focus to constraints (and the IVPs that implement them) that are expressable as boolean formulas. More complex expressions exist (*e.g.*, relations that are equivalent to a Turing Machine description) and we defer their study to future work, partly because boolean formulas are powerful enough for the purpose of expressing data structure constraints, and partly because we suspect the automatic synthesis of *correct* RPs from TM-equivalent IVPs is undecidable. The model supports manual synthesis and certification of these RPs.

For the class of IVPs that we do consider, each IVP provides a boolean condition that was violated. At this level of complexity, self-healing may be accomplished by finding a satisfiable assignment for that boolean condition in addition to any user-specified recovery actions. Therefore, synthesizing a list of RPs from the list of constraints on each IVP is achievable. While the Satisfiability problem is NP-Hard, it is at least decidable. We survey current software systems to get an idea of the complexity of typical assertion constraint conditions. A large portion of these applications do not use complicated expressions, and so the time for determining a satisfiable assignment of values to the expression atoms should be short. Finding appropriate values is driven by the definition of the constraints on the relevant CDIs. Here, however, the system must follow some heuristics (if completely automated) or human advice (if the RP is manually specified).

2.4 General Caveats

We summarize the challenges for our system here and address some more fully in Section 5.

1. **Synthesizing Policy**: *Doesn't the programmer have to write recovery policies ahead of time? Why not write the software correctly in the first place?* We consider automatic synthesis in Section 2.3. In cases where repair policy is synthesized by a human, SoD implies that the programmer shouldn't specify the recovery policy; the end-user or system administrator may actually provide a better specification. Furthermore, the goal of a repair policy is not necessarily aimed at fixing the underlying vulnerability. Instead, it should supply a configurable response that augments the wide variety of protection mechanisms available to serve as a short-term "band-aid."

2. **Exception Handling**: *Exception handling can be considered a type of repair policy that is synthesized before system deployment by a human.* While exception handling is a valuable tool for handling system errors, it lacks a number of properties that make it suitable as an automatic repair policy mechanism.

3. **Hiding Errors** *Doesn't the process of self-healing "hide" errors? If so, security is decreased because a broken system is being run without the knowledge of the administrator.* Self-healing does not preclude the system from reporting errors, alerts, or policy violations so that a human effort to address the source of the vulnerability parallels the automatic process, and CW has strict audit requirements.

4. **State Delegates**: *Boolean expressions form the basis of one class of recovery policy constraints. However, such expressions may only represent a measurement of a high-level property of another data structure or CDI.* We term such a relationship a "state delegate." For example, a CDI named `num_sorted` may be constrained to be less than the value `10`. Repair likely involves not only modifying the value of the CDI `num_sorted`, but also the underlying CDI. We defer the investigation of automatically synthesizing repair policies for these types of algorithmic relations.

5. **Nondeterministic Effects**: *Attempts to sandbox an application's execution must sooner or later allow the application to deal with global input and output sources and sinks that are beyond the control of the sandbox.* Repair attempts may fall short in situations where an exploit on one machine (*e.g.*, an electronic funds transfer front-end) that is being "healed" have visible effects on another machine (*e.g.*, a database that clears the actual transfer). Such situations require additional coordination between the two systems – they must logically be considered the same, and our procedures must be transactions that span both machines. Placing the machines in the same administrative domain somewhat amerliorates this challenge. In contrast, if a browser exploit caused a PayPal transaction to be initiated, a self-healing system may be able to recover control on the local machine, but the user would not have an automated recourse with the external PayPal system.

2.4.1 Model Limitations

Clark-Wilson (and our extension) rely on a number of assumptions. Perhaps the most significant is that each user is only permitted to execute a specific set of programs (TPs). The system should ensure that it is not possible for a user to augment their set of programs to bypass the SoD rules. The phenomena of emergent properties makes this assumption notoriously difficult to guarantee, because future configurations of the system may contain programs that can be combined in unexpected ways. Therefore, system reconfigurations must be recertified. This requirement is somewhat unrealistic for current and foreseeable software systems,

and complexity often makes the certification problem intractable, even though some utilities such as the Unix `sudo` program seek to provide just this type of enforcement.

More to the point, this security model relies heavily on the notion of authenticated principals with non-overlapping authorization roles. However, this requirement has the largest effect on security concerns, since, as the authors note, it relies on certification rules. Translating the model to real software is a challenge, especially since the complexity of modern systems threatens to violate many of the underlying security expectations. In our implementations, we make a number of reasonable design choices to illustrate the feasibility of the key concepts.

3 Implementation

We are constructing a prototype of our model for two different language systems: Java Virtual Machine bytecode and *x86* machine code. Our main reason for doing so was to demonstrate the general applicability of the model, and choosing both a "managed runtime" platform and a pure binary platform helps. Also, the application of the model to these systems shows that we are interested in enforcing higher-level constraints than those typically associated with buffer overflows. Finally, the prototypes demonstrate our ability to insert policy into program binaries and avoid cluttering the source code with calls to policy routines or having to recompile the application. Our Java implementation leverages the BCEL[4] Java bytecode parsing library and the Java Instrumentation Framework introduced in Java 5. Our *x86* prototype uses the PIN dynamic binary rewriting platform [19].

Our implementation is similar to aspect oriented programming: we insert calls to an inline reference monitor at well-defined boundaries (function exit). The IRM assess the state of the application with respect to the policy, and reconciles that state (*i.e.*, invoke the appropriate RP and RVP) if it violates policy (*i.e.*, the IVP returns "INVALID"). Figure 2.2 illustrates a psuedocode version of the IRM that we insert at appropriate places in an application's execution. The values of the IVP, RP, and RVP are dynamically determined by the instrumentation engine from the policy. Policy is specified as an XML file or as flat file, depending on the prototype, and contains CDI definitions and lists of relations as specified in Rules **E2**, **E5**, and **E6**. If the IVP, RP, and RVP functions are not already part of the codebase, they are dynamically inserted. To demonstrate the proof of concept, we leverage existing application code to provide the RPs rather than automatically synthesize them from the corrosponding IVPs.

Like SELinux policies, default repair policies can be written and distributed by application vendors, stakeholders, or competitors. Users can be presented with a menu option that presents recommended actions (vendor-certified) along with other more risky repair behaviors (*e.g.*, the repair choice may be the most comprehensive, but if some memory areas have been corrupted, it may only succeed with a 45% chance). Our model admits a repair quality scale as measured by an RVP.

4 Evaluation

The main purpose of our evaluation is to assess the hypothesis that automatic synthesis of RPs from boolean-expression IVPs is a feasible undertaking. We also want to confirm that the insertion of policy into a real software system is effective in providing the user with a customizable set of "repair" actions. For this latter test, we use Firefox 1.5 and the memory corruption vulnerability described in Mozilla Foundation Security

[4] `http://jakarta.apache.org/bcel/index.html`

Table 1: *Assertion Statistics.*

app	ta	tma	eqa	exa	lga	fna	tfa	tsa
cvs-1.11.22	245	194	133	86	53	10	4	0
gaim-1.5.0	5	81	1	4	0	0	0	0
httpd-1.3.36	248	53	153	30	74	57	11	0
james-2.2.0	115	129	84	54	17	54	1	0
mplayer-1.0pre8	849	166	500	95	409	107	40	0
openssh-4.3p2	7	69	0	0	7	0	0	0
qemu-0.8.1	120	35	81	3	33	5	34	0
sendmail-8.13.7	52	147	8	6	33	4	7	0

Table 2: *Assertion Statistics (cont.).*

app	cmx	$1a$	$2a$	$3a$	$4a$	$5a$	$g5a$
cvs-1.11.22	1.167	216	19	8	2	0	0
gaim-1.5.0	1	5	0	0	0	0	0
httpd-1.3.36	1.10	224	24	0	0	0	0
james-2.2.0	1.347	95	8	7	2	3	0
mplayer-1.0pre8	1.214	715	96	28	10	0	0
openssh-4.3p2	1	7	0	0	0	0	0
qemu-0.8.1	1.25	100	13	4	3	0	0
sendmail-8.13.7	1.038	50	2	0	0	0	0

Advisory 2006-04. After the vulnerability condition is triggered, we provide the user with the choice of clearing private data from the browser or saving all tabs.

4.1 Complexity of Assertions

In order to analyze the complexity of the assert expressions we performed a survey on widely-used open source applications written in both C/C++ and Java. The survey characterizes the complexity of assert expressions encountered in each application by a number of properties summarized in Table 3.

The results agree with our initial intuition: assertion expressions do not have a high degree of complexity. The average number of the atoms (an atom is defined as an operator and its arguments) per assert is less than 2. There are rare instances of expressions with 2 to 5 atoms. We found no higher grades in any of the analyzed applications. This implies that the atoms are easily identifiable and easy to manipulate. If we look at the structural detail of an atom, we can observe that the most common ones are based on the equality operator: an atom that is relatively easy to satisfy. The same simple strategy can be adopted for existential and logical expressions. The frequency of the assertions that include function calls is not high, as shown in Table 1 and Table 2, which is encouraging, because this type of expression is more complex to fix. Some of the applications show a number of assertions that are unequivocally meant to fail; they confirm our inituition from Section 1 that directly leveraging assertions may be the wrong approach. We performed a brief analysis of other applications, such as `valgrind-3.2.0` and `httpd-2.2.2`, due to the high volume of assertion expressions. Again we observed the same types of expressions, showing low complexity with the dominant

Table 3: *Key for Table 1 and 2*

app	the application under test
ta	total number of assertion-like statements
tma	total number of assertion-like statements in docs
eqa	number of equivalence assertions
exa	number of existential assertions (*e.g.*, x != NULL)
lga	number of logical assertions (*e.g.*, $x < 5$)
fna	number of assertions involving a function call
tfa	number of assertions trivially meant to fail (*e.g.*, 1!=1)
tsa	number of assertions trivially meant to succeed (*e.g.*, 0==0)
cmx	average complexity of assertions, measured in # of atoms
1a	total number of 1-atom assertions
2a	total number of 2-atom assertions
3a	total number of 3-atom assertions
4a	total number of 4-atom assertions
5a	total number of 5-atom assertions
g5a	total number of 6-atom and greater assertions

expressions being 1-atom assertions. This result is encouraging since it confirms our intuition that the time for determining a satisfiable assignment of values is small.

5 Discussion

This section considers some of the important challenges and related areas of work that we plan to consider more fully in the future, including automatic identification of instrumentation points and comparing our approach with exception handling.

5.1 Identifying Instrumentation Locations

Identifying locations in program code to insert calls to the reference monitor or policy evaluator is a challenging task. Many approaches to this problem exist, ranging from complete instrumentation to selective, retroactive insertion of specific checks. This range reflects the tradeoff between the level of protection for a system and the impact on system performance. Depending on the protection mechanism, IVP invocation could occur after every machine instruction, after every basic block, or after some specific class of machine instructions (*e.g.*, control-flow transfer, arithmetic, logic, floating point, *etc.*). Locations at higher levels of abstraction are also plausible places to insert policy checks. For example, the Java policy access control mechanism relies on calls to the `AccessController` being interwoven throughout the standard classes of the Java library. The approach we take in this paper is to remain flexible: only TPs specified in the policy are instrumented with calls to their appropriate IVP.

This flexibility naturally raises the question of which TPs to include in the policy specification. Should the TPs consist only of functions that have a known vulnerability? In this case, why not simply fix the vulnerability in the source and redeploy the application? How does one even know to include a TP in the policy if it contains an undiscovered vulnerability?

"Fixing" the source or waiting for a patch is beyond the scope of the problem we are considering. Protection mechanisms already exist that detect large classes of attacks. We focus on the complimentary issue

of what should be done when any attack is detected. A policy should apply to TPs containing both known and unknown vulnerabilities. The collection of TPs can be specified manually or automatically. Manual specification (by program authors, system administrators, or end-users) of a repair policy likely provides a high-quality, detailed response, but human time and error can influence its correctness. Automatically identifying TPs can be orders of magnitude faster, especially if a TP is identified in response to an exploit attempt. A completely automatic process needs some initial classification or training, but as with static analysis that identifies vulnerabilities in the first place, still lacks completeness. Ganapathy *et al.* [14] suggest an interesting approach. After a small collection of enforcement points are manually specified, their system synthesizes "fingerprints" of these code sequences and automatically identifies other locations in the program that match the fingerprint. These locations are candidates for instrumentation. In general, our approach does not require that the author of a repair policy know that a TP has a vulnerability: likely candidates may be selected according to heuristics or real evidence that a vulnerability exists, such as alerts from an IDS. Another possible approach may be to have the enforcement mechanism remember a snapshot of the application environment when attacks *do* occur: this state is useful as future evidence. Follow-on work could build a system that learns when environment conditions are ripe for attack. The recreation of those conditions could trigger the enforcement mechanism.

5.2 Exceptions Considered Unhelpful

We have elsewhere alluded to the relationship between our repair policy model and exception handling as it is implemented in modern programming languages like C++, Java, and C#. Even when employed, its use as a recovery mechanism is hampered by the programmer being unaware of the exact state of the system and the condition that has been violated. She is thus unable to effectively enumerate the possible responses or specify a decision module to choose among them.

Exception handling is a version of our policy where SoD is violated: the author and certifier of a RP is most likely the author of the TP it cleans up after. Exceptions themselves are little more than a signal from some IVP. As such, they often describe the symptom experienced, and not the root cause of the problem. Consider a `try...catch` clause that surrounds a number of statements dealing with arrays. At best, an exception due to an out-of-bounds array access may provide the programmer with a string and stacktrace indicating the line where the violation occurred.

However, there is little *programmatic* support in the exception itself for discovering which array and memory offset violated the constraint. Furthermore, if the handler were not in the same scope as the origin of the exception, the exception would need to marshal and unmarshal this locally scoped data into and out of the exception object instance. Without this type of information, responses can be nothing *but* manually specified. Another issue is that the programmer's (and thus the system's) awareness of the error begins at the moment that the exception is caught. This point in control flow may be in a completely different scope and level of abstraction.

```
try{
  int x = Integer.parseInt(s);
}catch(NumberFormatException n){
  log("x was not expected value, setting to default");
  x = DEFAULT_X_VALUE;
  s = ""+x;
}
```

Figure 3: *Handling a "Simple" Exception.*

Even for this basic of violations, it is not clear to a language designer what the best reaction should be – nor is this ideal, as the language should be a general tool. The language designer should refrain from injecting themselves into the application design process by forcing a one-size-fits-all approach onto applications written in the language. It is also not clear to the programmer what the best strategy is. She may not be sure how far the system has drifted off course after a number of functions and statements in the same scope as the exception have already executed. Therefore, the programmer really doesn't know about how to go about designing, much less writing a robust response to handle any particular exception. There are of course other "simple" cases, as in the example of Figure 5.2. Even this example requires a non-trivial amount of planning.

A developer must know that `parseInt()` will actually throw an exception. She must know that `s`'s content comes from an untrusted source (she might otherwise catch and ignore the compiler-mandated placement of the handler). There must exist a well-known default value for `x`. Each of the three statements in the handler reveals dependencies on the rest of the codebase. The syntax and semantics of the system's logging infrastructure must be known. The developer must create an informative error message. Finally, she must consider whether or not to alter the original input or let other code handle the illegal value on its own. As a result of these hurdles for even simple exception handling, we encourage efforts to augment exception generation with data that allows the system to self-heal, or at least gives the code author programmatic access to the state involved in the exception. Absent these capabilities, our repair model can provide a way to self-heal without having to modify program code.

6 Related Work

Nearly twenty years have passed since the publication of Clark and Wilson's description of a model for information integrity. Since then, it has become clear that the model also applies to execution integrity. In this section, we highlight the ways in which recent research reflects various portions of the Clark-Wilson model. We cover the major points of the Clark Wilson model elsewhere (Section 2.1).

6.1 Intrusion Reaction

Intrusion defense mechanisms typically respond to an attack by terminating the attacked process[5]. Even though it is considered "safe", this approach is unappealing for a variety of reasons. Crashing leaves systems susceptible to the original fault upon restart and risks losing accumulated state.

Effective remediation strategies remain a challenge, but some first efforts include failure oblivious computing [24], STEM's error virtualization [26], DIRA's rollback of memory updates [27], crash-only software [6], and data structure repair [10, 11]. While most of these techniques can be implemented as part of a RP, they all have shortcomings. The first two may cause a semantically incorrect continuation of execution (although the Rx system [23] attempts to address this difficulty by exploring semantically safe alterations of the program's environment until execution succeeds). DIRA's logging may be too expensive for certain environments. Crash–only software is a unique approach to the problem, but requires that systems be re-written to use the services advocated by that approach. The notion of data structure repair is closest in spirit to our approach.

Oplinger and Lam propose [22] using hardware Thread-Level Speculation (TLS) to improve software reliability. Their key idea is to execute an application's monitoring code in parallel with the primary computation and roll back the computation "transaction" depending on the results of the monitoring code. In an

[5]Some attempt to generate either vulnerability or exploit signatures as well. See [18, 27, 30].

interesting twist on detection, the ReVirt system employs virtual machine logging to work backward from vulnerability predicates to discover the point of infection [15].

6.2 Protecting Control Flow

Work in assuring that the execution of a process does not go astray has typically focused on protecting the integrity of a process's jump targets. For example, StackGuard [9, 13] and related approaches attempt to detect changes to the return address (or surrounding data items) of a stack frame. If the integrity of this value is violated, execution is halted. These approaches quite naturally fit the Clark-Wilson Integrity Model: the CDI is the return address in the current frame, the TP is the procedure for popping a stack frame, and the IVP is the procedure that these systems add to verify that the CDI is still valid. What these systems lack is the notion of both a Repair Procedure and a Repair Validation Procedure. Instead, they can be considered to have a vestigial RP: optionally logging the fault and then terminating the process.

Starting with the techinque of *program shepherding* [17], the idea of enforcing the integrity of a process's control flow has been increasingly researched. Program shepherding validates branch instructions in IA-32 binaries to prevent transfer of control to injected code, and to make sure that calls into native libraries originate from valid sources. Abadi *et al.* [1] propose formalizing the concept of Control Flow Integrity, observing that high-level programming often assumes properties of control flow that are not enforced at the machine language level. CFI provides a way to statically verify that execution proceeds within a given control-flow graph (the CFG effectively serves as a policy). The use of CFI enables the efficient implementation of a software shadow call stack with strong protection guarantees. CFI complements our approach in that it can enforce the invocation of the IVP, RP, and RVP procedures (rather than having malcode attempt to skip past these checks). The integrity model we suggest encompasses both data and control flow.

Control flow is often corrupted because a UDI enters the system and is eventually incorporated into part of an instruction's opcode, set as a jump target, or forms part of an argument to a sensitive system call. A great deal of work has focused on dataflow analysis of tainted data and ways to prevent such attacks from succeeding [28, 21, 20, 8]. Bhatkar, Chaturvedi and Sekar's [3] work on dataflow anomaly detection examines the temporal properties of data flow through a sequence of system calls in order to detect intrusion attempts. One benefit of this model of dataflow integrity is that temporal properties enable formal reasoning about the security properties of a program.

6.3 Policy Enforcement

Since security is often delegated to a secondary concern in the production of software systems, many researchers have undertaken work that attempts to retrofit the enforcement of security policies on these codebases. The work by Ganapathy *et al.* [14] is most closely related to ours, and it discusses ways to insert authorization checks into the X server. They face many of the same issues that we do, including identification and placement of calls to the policy evaluator or reference monitor and specifying the content of the policy. Our work differs in two important aspects. First, we focus on integrity policy and how to automatically repair violations of that policy. Second, our system works by dynamically instrumenting programs binaries and does not require source code to operate (although more precise policies can be constructed with knowledge of source-level names and objects).

Identifying appropriate program locations to instrument is an important consideration discussed in Section 5.1. In the literature, examples of "complete" protection include the techniques advocated by StackGuard. Similarly, array-bounds checking in Java ensures each array access falls within the limits of the array. Complete process emulation is even more expensive and employed in approaches like Instruction Set

Randomization [2, 16] and binary-level taint-tracking [8, 21]. Taint-based data flow analysis is equivalent to tracking Unconstrained Data Items (UDI) through the system until they affect the control flow of a TP.

ASSURE [25] proposes the notion of *error virtualization rescue points*. A rescue point is a program location that is known to successfully propagate errors and recover execution. The insight is that a program will respond to malformed input differently than legal input; locations in the code that successfully handle these sorts of anticipated input "faults" are good candidates for recovering to a safe execution flow. ASSURE seeks the best location to which to "teleport" a failure that is detected elsewhere in the program.

7 Conclusions

The underlying thesis of this paper is that integrity can support availability. In many environments, users value continued execution rather than the abrupt process termination that serves as the widely accepted default response to an attempted intrusion. Even if a system has automatic response capabilities, system security is often a matter of *policy*; systems need flexibility to remain useful in a variety of evolving environments. In this paper, we extend the Clark-Wilson integrity model with mechanisms for specifying and enforcing repair procedures and apply this updated model to the problem of execution integrity.

The goals of our future work are mainly technical in nature. We will address the challenge of applying rollback and replay to our system. This capability would allow us to to speculate the policy constraints (IVPs) and repairs (RPs and RVPs) to avoid altering the execution environment unless repair succeeds. In addition, we will continue to assess how the system can best be scaled (*e.g.*, applying policy updates and resolving policy conflicts). One interesting area of study may be to have the system learn what the best response is over time; that is, what sequence of RPs are "best" to fix a set of invalid CDIs.

Program designs will always lack a complete description of how to handle all errors. The opportunity and motivation to take advantage of these errors will not disappear as long as computing systems are trusted with the task of processing, transmitting, and storing important data. No system can be perfectly secure, but we can provide well-formed recovery mechanisms and invoke them automatically. An integrity repair model assists in bridging the gap between current systems and systems that are able to automatically self-heal.

A Clark-Wilson Model Rules

1. **C1:** All IVPs must properly ensure that all CDIs are in a valid state at the time the IVP is run.
2. **C2:** All TPs must be certified valid. For each TP and each set of CDI that it may manipulate, the security officer must specify a "relation" which defines that execution. A relation is of the form: (TP_i, (CDI_a, CDI_b, CDI_c, ...)), where the list of CDIs defines a particular set of arguments for which the TP has been certified.
3. **E1:** The system must maintain the list of relations specified in rule C2, and must ensure that the only manipulation of any CDI is by a TP, where the TP is operating on the CDI as specified in some relation.
4. **E2:** The system must maintain a list of relations of the form: (USERID, TP_i, (CDI_a, CDI_b, CDI_c,...)), which relates a user, a TP, and the CDIs a TP may reference.
5. **C3:** The list of relations in E2 must be certified to meet the separation of duty requirement.
6. **E3: (Authentication)** The system must authenticate the identity of each user attempting to execute a TP.
7. **C4: (Auditing)** All TPs must be certified to write to an append-only CDI with all information necessary to permit the nature of the operation to be reconstructed.
8. **C5: (Taint)** Any TP that takes a UDI as input must be certified to convert a UDI to a CDI, or reject the UDI, for any possible value of the UDI.

9. **E4: (Separation of Duty)** Only the agent permitted to certify entities may change the list of such entities associated with other entities: specifically, that associated with a TP. An agent that can certify an entity may not have any execute rights with respect to that entity.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[3] S. Bhatkar, A. Chaturvedi, and R. Sekar. Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 105–120, August 2003.

[5] A. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to dependability. In 10^{th} *ACM SIGOPS European Workshop*, Saint-Emilion, France, Sept. 2002.

[6] G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, May 2003.

[7] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[8] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, 2005.

[9] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Security Symposium*, 1998.

[10] B. Demsky and M. C. Rinard. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the* 1^{st} *Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[11] B. Demsky and M. C. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, October 2003.

[12] G. W. Dunlap, S. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, February 2002.

[13] J. Etoh. GCC Extension for Protecting Applications From Stack-smashing Attacks. In *http://www.trl.ibm.com/projects/security/ssp*, June 2000.

[14] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting Legacy Code for Authorization Policy Enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[15] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, 2005.

[16] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 272–280, October 2003.

[17] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

[18] Z. Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2005.

[20] J. Newsome, D. Brumley, and D. Song. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13^{th} *Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.

[21] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12^{th} *Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[22] J. Oplinger and M. S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the* 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

[23] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, 2005.

[24] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[25] S. Sidiroglou, O. Laadan, J. Nieh, and A. Keromytis. ASSURE: Autonomic Software Self-Healing Using Error Virtualization Rescue Points. Unpublished manuscript, 2006.

[26] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

[27] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the* 12^{th} *Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[28] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the* 11^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, October 2004.

[29] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the* 22^{nd} *International Symposium on Reliable Distributed Systems (SRDS)*, 2003.

[30] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.