

Avoiding Serialization Vulnerabilities through the Use of Synchronization Contracts

Reimer Behrends, R. E. K. Stirewalt, and Laura K. Dillon
{behrends,stire,ldillon}@cse.msu.edu
Department of Computer Science and Engineering
Michigan State University

Abstract. *Synchronization contracts facilitate the development and analysis of multi-threaded programs and can be used to guard against serialization vulnerabilities, which pose serious security risks and which are very difficult to detect. In practice, however, real applications cannot be written entirely with a language that supports synchronization contracts, but must incorporate system libraries and third-party code. This paper describes a technique for removing serialization vulnerabilities from existing code in source or (with linker support) binary form, thereby permitting the code to be safely integrated into an application that is written using synchronization contracts. We have applied the technique in writing a multi-threaded web server using synchronization contracts.*

1. Introduction

In classifying security flaws, Landwehr et al. [18] distinguish one class, called *serialization vulnerabilities*,¹ that is unique to multi-threaded systems. In this class, they include all “flaws [in multi-threaded systems] that permit the asynchronous behavior of different system components to be exploited to cause a security violation.” Serialization vulnerabilities can give rise to so called *time-of-check-to-time-of-use* (TOCTTOU) errors, in which one process validates a condition that enables some operation and another process asynchronously invalidates this same condition before the first process performs the operation. For example, operations in asynchronous processes might check that a bank account has a positive balance, enabling both processes to execute withdrawal operations, although the first withdrawal operation that occurs may in fact empty the account. Other types of serialization vulnerabilities may enable an attacker to access data for which it does not have necessary security clearance, such as the passwords to accounts that the attacker does not hold. Unfortunately, serialization vulnerabilities are notoriously difficult to detect [1]. They frequently go unnoticed because they often do not result in errors and when they do, the errors are not systematically flagged.

Synchronization contracts can guard against two major sources of serialization vulnerabilities in a multi-threaded OO system—namely, inadequate synchronization logic and execution races enabled by aliasing.² In essence, synchronization contracts permit the problem of detecting serialization errors, a difficult problem, to be recast into that of detecting boundary condition errors [18], a much easier one. Serialization errors manifest as detectable violations of the boundaries of data spaces, which

¹termed *serialization errors* by Du and Mathur [12] and *synchronization faults* by Aslam [1].

²two or more processes asynchronously accessing the same data object through references that they both hold.

are associated with processes through synchronization contracts. Such violations result in observable run-time exceptions.

However, it is seldom practical and it is sometimes impossible to write a complete multi-threaded system entirely in a language that supports synchronization contracts. For example, a web server will need to use system libraries and interface with third-party code, which will not have been written using synchronization contracts. We say that such code and the languages in which it is written are *contract-unaware*. In contrast, we dub a language that supports synchronization contracts and code written in such a language as *contract-aware*. While systems written in contract-aware languages may be safe from serialization vulnerabilities, no guarantees can normally be made about any contract-unaware code that they interface with. This paper describes a technique for removing serialization vulnerabilities from existing code in source or (with linker support) binary form, thereby permitting contract-unaware code to be safely integrated into a contract-aware application. We have applied the technique in writing a contract-aware web server.

The rest of this paper is structured as follows. Section 2 describes our model of synchronization contracts, showing how they are used to recast the problem of detecting serialization vulnerabilities into that of detecting boundary condition errors. Sections 3 and 4 deal with the problem of protecting modules in languages not equipped with synchronization contracts (e.g., legacy C code and system libraries) against serialization vulnerabilities. Section 5 then validates our approach by applying it to the example of a multi-threaded web server. Section 6 concludes with a discussion of related work.

2. Synchronization Contracts

Our approach to specifying and automatically implementing security requirements in concurrent systems builds on Meyer’s *design by contract* method, in which a developer reasons about the safety of component interactions using *contracts*, which specify the rights and responsibilities of clients and suppliers [20]. While originally designed for sequential systems, researchers have recently looked to extend and articulate the design-by-contract approach to aid in the design and verification of concurrent systems [5]. Contracts used for this purpose are called synchronization contracts. In our model of (synchronization) contract-aware components, a run-time system dynamically negotiates concurrent interactions to guarantee safety requirements, specifically mutual exclusion and freedom from certain kinds of deadlock and starvation [3, 4, 23]. This section provides brief overviews of this model (Section 2.1) and of a web-server application that we developed in our contract-aware version of Eiffel and that we will use in examples in the rest of the paper (Section 2.2).

2.1. Our Contract Model

In our model, processes operate in disjoint data spaces, called *realms*, that comprise one or more *synchronization units*. A synchronization unit is a cohesive collection of objects that a process will need to access together (as a “unit”). Every object created during execution of a program belongs to a unique synchronization unit over the course of its lifetime, and it is illegal for a process to access data in a unit³ that is not in its realm. Suppose, for example, that when executing within some unit u , a process attempts to access data in some unit v . Then u is necessarily in the realm of the process and, if v is in this same realm, the access succeeds; otherwise a run-time exception is raised. Because realms are disjoint, it is not possible for two processes to concurrently access the same memory location. As

³for brevity, we often refer to a synchronization unit as just a unit. Similarly, we often refer to synchronization contracts (respectively concurrency constraints) as just contracts (respectively constraints).

a program executes, shared units may *migrate* from one realm to another, but these units can never be accessed simultaneously by different processes. This migration is governed by contract negotiation, which we now explain.

Each unit invokes services of other (supplier) units according to a *concurrency constraint* that specifies the conditions under which the former assumes it has exclusive access to the latter. The methods that implement the functionality of a unit are *designed by contract*, meaning the code is written assuming that the unit has exclusive access to supplier units under the conditions specified by the unit's concurrency constraint. Thus, a unit is an example of a contract-aware component, where the contract is defined by a concurrency constraint. In a multi-threaded web server, for example, each thread may contain a unit called a *request handler*, which is responsible for handling and serving resource requests. The server will also contain an *authenticator* unit, which a request-handler unit will require access to only under certain conditions, e.g., when the handler is trying to authenticate the requester. The request handler's concurrency constraint codifies these conditional dependencies, and the unit itself maintains a set of *condition variables*, which record the state of the unit's computation, thereby making the conditions observable by a run-time system that negotiates and enforces contracts.

We say that a unit's contract has been *negotiated* if those supplier units, which the unit's concurrency constraint guarantees are accessible, are actually in the unit's realm. Because the makeup of this set of supplier units depends on the values of condition variables, previously negotiated contracts may need to be *renegotiated* when the values of condition variables are modified. Such renegotiation involves the migration of units among realms. If one or more of the suppliers that are needed to satisfy a unit's concurrency constraint cannot be migrated into a unit's realm—e.g., if they are currently in use by other processes—then the contract is said to be *pending*. Our model dictates that a process must block if its realm contains any units with pending contracts. A run-time system renegotiates contracts in response to changes in condition variables, thereby migrating units among realms, scheduling some processes and suspending others.

In controlling migration of units and scheduling of processes, the run-time system guarantees the invariance of the concurrency constraints in a program, thereby enforcing the contracts. This functionality is achieved without the programmer needing to write code that explicitly manipulates realm, unit, or thread representations. Instead, the programmer writes code that maintains condition variables to encode and reflect the current “state” of the computation and constraints that declaratively specify the suppliers that a unit assumes it may access in each of these states. We contend that such code—by virtue of being local to a single unit—and concurrency constraints—by virtue of being declarative and expressed at appropriately high levels of abstraction—are less susceptible to synchronization errors than the typical code that a programmer writes to explicitly synchronize processes that share data. Moreover, errors due to code that does not conform to the stated contracts are easily detected at run-time by an efficient realm-boundary check. When one unit attempts to access another, the run-time system checks that the units belong to the same realm, allowing the access only if the check succeeds and raising a run-time exception otherwise. This latter mechanism permits the problem of detecting serialization vulnerabilities to be recast into that of checking for violations of a realm boundary.

In prior work, we extended the Eiffel language [19] with concurrency constraints [3] and showed how concurrency constraints can be integrated into other object-oriented languages [4, 23]. In this extension, developers write contract-aware programs by defining one or more *synchronization classes*, which are instantiated to produce synchronization units, and adorning each of these classes with a *concurrency clause*, which defines the structure of the associated concurrency con-

```

1 synchronization class REQUEST_HANDLER inherit PROCESS_BASE
2 feature { NONE } -- instance variables
3   auth_stage, content_stage: BOOLEAN      -- condition variables
4   authenticator: AUTHENTICATOR           -- unit reference
5   connection: CONNECTION_SOCKET         -- unit reference
6   content_dispatcher: CONTENT_DISPATCHER -- unit reference
7   ...
8 feature { ANY } -- operations (methods)
9   authenticate ( request : REQUEST ) is
10  do
11    auth_stage := true
12    authenticator.validate(request)
13    auth_stage := false
14  end
15  ...
16 concurrency -- concurrency clause follows
17  connection
18  auth_stage => authenticator
19  content_stage => content_dispatcher
20 end -- class REQUEST_HANDLER

```

Figure 1. Elided definition of the synchronization class REQUEST_HANDLER

straints. Figure 1 depicts part of an example synchronization class, which is taken from the web server application described in the next section. The keyword `synchronization` (line 1) signifies that class `REQUEST_HANDLER` is a synchronization class. The class declares several instance variables, some of which are condition variables (e.g., `auth_stage` and `content_stage` in line 3) and some of which reference other units (e.g., `authenticator`, `connection`, and `content_dispatcher` in lines 4–6).⁴ The concurrency clause appears after the keyword `concurrency`; it contains three conjuncts, shown in three separate lines. The first conjunct (line 17) declares that whenever a `REQUEST_HANDLER` unit executes it assumes it has access to the the unit, if any, referenced by its `connection` attribute. By contrast, the next two conjuncts (lines 18 and 19) say that the unit assumes it has access to the unit referenced by the `authenticator` (respectively `content_dispatcher`) only when the condition variable `auth_stage` (respectively `content_stage`) is true.

2.2. Web-Server Example

In the sequel, we will refer extensively to our web-server implementation, whose design is depicted in Figure 2. The diagram follows the standard UML notation, but with the concurrency clause of a synchronization class rendered in curly braces adjacent to the box. The server works as follows.

⁴In our Eiffel extension, a `BOOLEAN` variable used in a concurrency constraint is inferred to be a condition variable, and an attribute whose declared type is a synchronization class is inferred to reference a synchronization unit. We elide the definitions of synchronization classes `AUTHENTICATOR`, `CONNECTION_SOCKET`, and `CONTENT_DISPATCHER`.

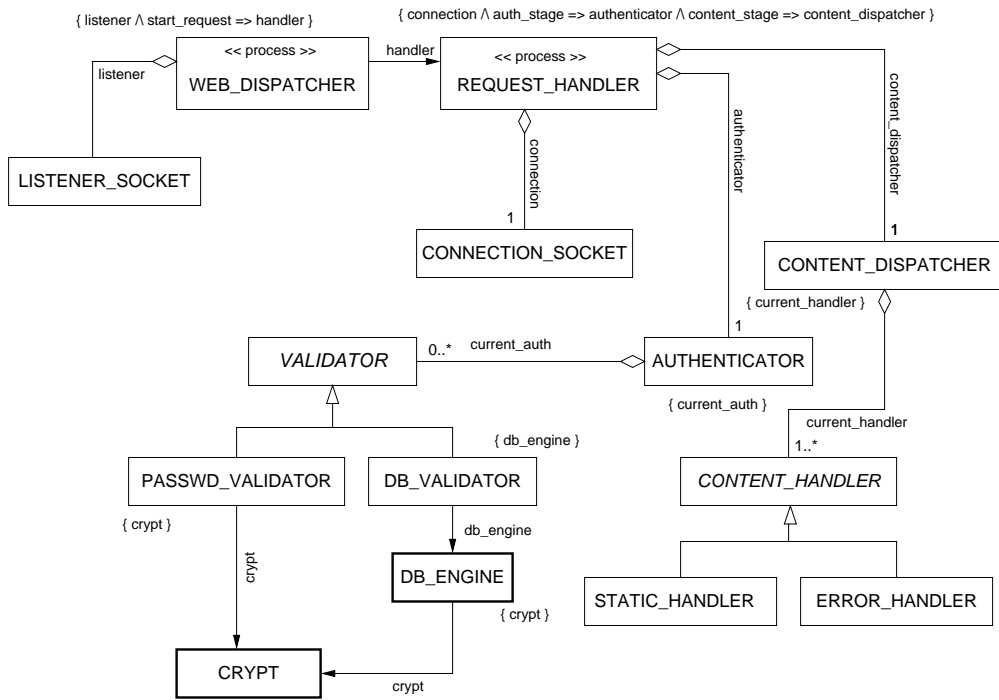


Figure 2. UML Class Diagram of a Web Server

Connections from web clients are received through a `WEB_DISPATCHER` unit, which accepts connections through a `LISTENER_SOCKET` unit. Upon accepting a connection, the `WEB_DISPATCHER` unit creates a `CONNECTION_SOCKET` unit, spawns a new process inside a `REQUEST_HANDLER` unit, and transfers the connection socket to the new request handler. The request handler then reads and parses a web request from the `CONNECTION_SOCKET` unit. Authentication is performed by an `AUTHENTICATOR` unit, which checks if a request requires authentication and then invokes several authentication schemes, one after the other, until the request is authenticated or all attempts have failed. Supported authentication units include a `PASSWD_VALIDATOR` unit, which uses a simple `crypt`-based authentication scheme, and a `DB_VALIDATOR` unit, which uses a database engine that is also based on `crypt`. The content generation stage, performed by a `CONTENT_DISPATCHER` unit, dispatches the request to one of two content handlers, a `STATIC_HANDLER` unit for static web pages or a `ERROR_HANDLER` unit for pages that could not be served.

3. Making Libraries Contract-aware

Applications developed in a contract-aware language are less susceptible to serialization vulnerabilities that arise from aliasing or faulty synchronization. In practice, however, many applications require the functionality of system libraries or third-party code, which are contract-unaware and thus may contain serialization vulnerabilities. For example, code generated by tools such as `lex` and `yacc` and generators for domain-specific languages, as well as many commonly used POSIX functions, such as `crypt`, `strtok`, and `gethostbyname`, suffer from serialization vulnerabilities. Our approach to safe integration is to (1) partition a contract-unaware library into a collection of modules, (2) associate with each module a synchronization unit for which contract-aware clients must negotiate in order to use the module's functions and data, and (3) instrument these modules with logic for coordinating with a synchronization unit. The main contribution of this paper is a technique for

instrumenting and then safely integrating contract-unaware modules into applications written in a contract-aware language while guarding against serialization vulnerabilities.

3.1. Libraries, modules, and units

To explain the details of our approach requires some new terminology. We use the generic term *library* to refer to system libraries and third-party software that an application programmer might wish to use but that is contract-unaware. A library comprises one or more functions and zero or more *static variables*, i.e., variables that are located in static memory, that are used by library functions, and that may or may not be visible outside the library. Generally speaking, serialization vulnerabilities due to faulty synchronization occur when two processes concurrently invoke library functions that manipulate these static variables, and vulnerabilities due to aliasing occur when a reference to one of these static variables escapes and thus can be directly accessed by the client application.

In contrast to a library, a *module* refers to a cohesive collection of functions and static variables in a library, such that no function in the module uses a static variable that is not also in the module, and every static variable in the module is only used by functions in the module. Link archives, e.g., `libc.a`, are examples of modules; however, a module could be a purely conceptual entity. For instance, within the POSIX library, we identified modules such as `crypt_module`, which contains the function `crypt` and the static variables it uses, and `name_services_module`, which contains name-service functions, such as `gethostbyname`, and the static variables used by these functions. Users of a contract-aware language access the functions and data of a module indirectly, by invoking operations on a special kind of unit, called an *external synchronization unit*. Because an external synchronization unit associates to an entire module, and because a unit may belong to only a single process at a time, we partition a library into fine-grain conceptual modules to maximize concurrency.

External synchronization units do not directly access functions and data in a conceptual module. Rather, they access so-called *decorator modules*⁵, which instrument the functions and data in a conceptual module with logic for checking whether an access crosses a realm boundary and for replicating shared data to prevent vulnerabilities due to aliasing. Given a conceptual module M , a decorator module, M -Decorator, is another module that exports the same interface as M , i.e., for each function f in the interface of M , M -Decorator declares a function whose signature is identical to that of f . Moreover, the definition of f in M -Decorator contains an invocation of the f in M . We refer to functions in M -Decorator as *function wrappers* and say that they *wrap* the corresponding function in M . Our approach uses function wrappers to instrument calls to library functions with logic that checks for realm-boundary violations and that replicates data so that no reference to a static variable in the library can escape a call to the wrapper.

3.2. Defining external synchronization units

Contract-aware clients use contract-unaware library functions and data by invoking the operations of an external synchronization unit, whose methods access function wrappers in a decorator module. For a client unit to access the functionality of some conceptual module, the client must first negotiate a contract for its associated unit. In the contract-aware program, each external synchronization unit is an instance of an *external synchronization class*, which is a special kind of synchronization class that: (1) can be instantiated at most once per program run, and (2) exports the same interface as the module that its singleton instance encapsulates. The first property insures that a one-to-one correspondence exists

⁵these modules are named after the *decorator pattern* [15], which indicates their purpose.

between shared modules and the corresponding units. Commensurate with the singleton pattern [15], client code may attempt to instantiate the class multiple times, but at most one instance will be created.

In our Eiffel extension, the keyword `external` designates an external synchronization class. For example, the following elided code snippet depicts the declaration of the external synchronization class whose instance provides contract-aware clients with access to the library function `crypt`:

```
external synchronization class CRYPT
...
  crypt( key, salt : STRING ) : STRING is
    external
    ... -- includes a call to the C function crypt.
  end
end
```

This class provides a method `crypt` whose interface conforms to that of the library function of the same name, except that the library function uses C-style strings whereas this method uses instances of the Eiffel class `STRING`. The method contains an *external clause*, which is used to include code from a foreign language into Eiffel programs. For brevity, we elide this code, most of which is concerned with translating between C-strings and Eiffel strings. External synchronization classes may also declare attributes, including references to other units, and a concurrency constraint.

An external synchronization class requires a concurrency constraint if its associated module uses the data or functions of another module. For example, in Figure 2, the external synchronization class `DB_ENGINE` declares a concurrency constraint involving the unit that plays a role in its `crypt` association. Consequently, the singleton `DB_ENGINE` unit will have a contract that references the singleton `CRYPT` unit.⁶ Notice that without a contract to guarantee that these units are in the same realm, a function in `db_engine_module`⁷ might invoke a function in `crypt_module` when the `CRYPT` unit is not in the same realm as the `DB_ENGINE` unit. Such a realm-boundary violation will be caught and flagged as a run-time exception.

3.3. Integrating external synchronization units and decorator modules

Having extended the contract-aware language with the ability to declare external synchronization classes, contracts may now refer to external units, which means the unit associated with a module will be accessible to at most one process at a time. As described previously, methods in an external synchronization class contain external clauses that allow the invocation of functions in C libraries. In this case, the calls will actually be to function wrappers in decorator modules. What remains is to design the decorator modules so that the function wrappers check for realm-boundary violations prevent references to static variables from escaping function invocations.

Function wrappers in the decorator module must check for realm-boundary violations when they are invoked. To perform these checks, each decorator module maintains a static variable that references its corresponding external synchronization unit. Function wrappers can then check for boundary violations by invoking a function in our run-time system, which checks if the given unit is in the realm

⁶In the diagram, bold boxes indicate that a synchronization class is external.

⁷i.e., the module that contains the library functions and static data used to support the database engine.

```

/* Configuration section. */
static sync_unit* external_unit;

void init_decorator(sync_unit* u)
{
    external_unit = u;
    Declare_Thread_Local_Vars;
    Other_Initialization;
}

/* Wrapper section. */
result_t _wrap_func(t1 arg1, ..., tn argn)
{
    result_t result;
    ASSERT_IN_REALM(external_unit);
    Deep_Import_arg1(arg1);
    ...
    Deep_Import_argn(argn);
    result = _real_func(arg1, ..., argn);
    Deep_Export_result(result);
    return result;
}

```

Figure 3. Decorator Module Template

of the calling process. Notice that it is necessary to check for boundary violations in the function wrapper, as opposed to in the method of the external synchronization unit, because a library function might be invoked from another library function in a different module. This design has the advantage that synchronization contracts will be enforced even when a library function is accessed from other contract-unaware code.

Vulnerabilities due to aliasing occur when a reference to a static library variable escapes an otherwise protected invocation of a library function. To guard against such vulnerabilities, decorator modules are designed to prevent such references from escaping the invocation of the function wrapper. More precisely, suppose a conceptual module contains a function f that returns a reference to a static variable v . The function wrapper that wraps f must return something other than v . Our wrappers work by copying the value of an escaping static variable (e.g., v) into a *thread-local variable*, references to which are always safe to return.

Briefly, a thread-local variable is a logical variable that manifests in independent storage locations local to each thread in the system. For example, suppose a program with three active threads— T_1 , T_2 , and T_3 —declares a thread-local variable $f_{\circ\circ}$. Then each thread can access $f_{\circ\circ}$, but the address of $f_{\circ\circ}$ will resolve to a different location depending on the accessing thread.⁸ Because thread-local variables resolve to different locations depending on the accessing thread, it is safe to allow a reference to such a variable to escape a function invocation. Thus, to prevent a reference to a static variable from escaping, a function wrapper performs a deep copy of the referent into a thread local variable and then returns a reference to the thread-local variable. We refer to such copying as *deep exporting*.

Obscure aliasing vulnerabilities can also occur through argument passing. If an external synchronization unit functions as a supplier, its operations can be invoked with a reference to data in the client unit. If this reference is passed to a library function, the function could store it in a static variable and later modify the client data by dereferencing this variable. To prevent such aliasing errors, we copy arguments into the static variables in the decorator module and then forward references to these copies to the library function. We refer to this process as *deep importing*.

4. Creating Decorator Modules

We now explain how to construct a decorator module. To aid in the explanation, Figure 3 depicts a generic template of a decorator module, implemented in the C language. In this template, bold items denote template parameters, which must be replaced with type names or uses of custom macros that the developer must write. To support developers in using this template, we provide a support library that defines the types and macros that are depicted in the figure and also some additional support functions that are useful in writing the custom code macros required to instantiate the template.

4.1. Brief introduction to support facilities

As discussed in Section 3.3, each decorator module must declare a static variable that references the external synchronization unit associated with the module. Our template refers to this static variable as `external_unit`. The support library provides two macros, `IN_REALM` and `ASSERT_IN_REALM`, for checking realm boundaries. Both of these macros check whether a given unit is in the realm of the currently executing process. `IN_REALM` expands to an expression that returns true if the unit is in the current realm and false otherwise. `ASSERT_IN_REALM` expands into a statement that raises a run-time exception if the given unit is not in the current realm. As shown in Figure 3, each function wrapper uses this assertion before importing any data or calling the library function.

Our library also provides facilities for declaring and accessing thread-local variables. The function `declare_thread_local_var` declares a new thread-local variable, returning a handle that is used to access it. In our implementation, all thread-local variables must be declared (by invoking this function) before any threads are created. To use the handle returned by `declare_thread_local_var`, we provide the macro `THREAD_LOCAL`, which maps a handle to a memory location.

4.2. A Template for Building a Decorator Module

A decorator module consists of two main sections, a *configuration section* and a *wrapper section*. The configuration section holds the functions used to initialize the decorator and the variables used to store the configuration information. The module will be initialized, calling the function `init_decorator`, at program start by the constructor of the associated external synchronization unit. Function `init_decorator` initializes the reference to the external unit and declares any thread-local variables that are needed to prevent the escape of references to static variables.

The wrapper section contains function wrappers, one for each function exported by the decorated module. Each function wrapper in the decorator will, at runtime, intercept calls to the corresponding wrapped function, enforce synchronization contracts, import arguments, call the wrapped function, and export the result. To wrap a function `func` with result type `result_t` and argument types `t1`, ..., `tn`, the synchronization decorator must contain a function `_wrap_func` with the same result and argument types. If the realm-boundary assertion succeeds, then the use of static variables in the function wrapper is now thread-safe. A deep import is performed for each argument, indicated by the use of the template parameters **Deep_Import_arg1** through **Deep_Import_argn**. The programmer must replace each of these template parameters with the code that performs the deep import. Next, the original version of `func` is called via a call to `_real_func`. Next, a deep export is performed on the result of the function, indicated by the use of the **Deep_Export** template parameter. The programmer

⁸Typically, each thread in a system maintains a pool of memory that cannot be accessed by other threads, and a thread-local variable is implemented as a uniform offset into these memory pools.

```

1 #include "tela.h"                20 /* Wrapper section */
2 #include <stdlib.h>              21
3                                  22 char *
4 /* Configuration section. */    23 __wrap_crypt(char *key, char *salt)
5                                  24 {
6 static sync_unit *crypt_unit;    25 char *result;
7 static int crypt_result;         26 static char *key_c, *salt_c;
8                                  27
9 char *__real_crypt(char*, char*); 28 ASSERT_IN_REALM(crypt_unit);
10                                  29 /* Importing arguments: */
11 void                             30 import_string(&key_c, key);
12 init_crypt_decorator(sync_unit *u) 31 import_string(&salt_c, salt);
13 {                                  32 /* Calling the original routine */
14 crypt_unit = u;                  33 result = __real_crypt(key_c, salt_c);
15 crypt_result =                    34 /* Exporting result: */
16     declare_thread_local_var(     35 if (result != NULL)
17     "crypt_result", ... );        36     result =
18 }                                  37     strcpy(THREAD_LOCAL(crypt_result),
                                   38         result);
                                   39 return result;
                                   40 }

```

Figure 4. Decorator Module for Crypt

must replace this template parameter with the code that performs the deep export or omit it if a deep export is not necessary.

Once a decorator module has been written and compiled, the resulting object code must be linked with the rest of the system such that symbolic references to wrapped functions are linked to the associated function wrappers. We accomplish this using the GNU linker, which provides a `--wrap` option that takes a function name as its argument. When supplied with the option `--wrap func`, the linker will redirect all calls of `func()` to calls of `__wrap_func`, and all calls of `__real_func` will be redirected to the original version of `func`. Consequently, when developing a wrapper for a function `func`, one must name the `__wrap_func` and invoke the wrapped function using the name `__real_func`.

Figure 4 illustrates how we instantiated the generic template to construct a decorator for the `crypt` module. The configuration part declares a variable (`crypt_unit`) that references the external unit (line 6), and an initialization function called `init_crypt_decorator` (lines 11–18). The initialization function first initializes `crypt_unit` and then declares a thread-local variable called `crypt_result` (lines 15–17).

The wrapper for function `crypt` is defined on lines 22–40. It begins with the check that the unit is part of the realm of the current process (line 28). Next, the arguments are imported (lines 30–31)⁹. Argument import uses an auxiliary function `import_string` (not shown in figure) that imports strings into static variables. Following the call to `__real_crypt` (line 33), the value of `result` must be exported to a thread-local variable (lines 35–38) which is then returned.

⁹Note that in the case of `crypt`, importing the arguments could be omitted, because `crypt` will not use the arguments after the function call returns. Therefore, omitting the import code would not introduce aliasing errors.

5. Validation: A Multi-threaded Webserver

Our approach attempts to enable contract-aware programs to safely use functions in contract-unaware libraries without incurring serialization vulnerabilities. To validate the feasibility of this approach, we extended a contract-aware language, in this case our Eiffel-language extension, and runtime system and applied it to create a multi-threaded webserver that uses a contract-unaware database engine and also the `crypt` routine. Our design, described in Section 2.2, closely models the design of multi-threaded versions of the popular Apache webserver, which is known to exhibit serialization vulnerabilities.¹⁰ Our design contains two external synchronization units, instances of `CRYPT` and `DB_ENGINE`. The `CRYPT` unit is accessed directly from instances of `PASSWD_VALIDATOR` and indirectly from instances of `DB_VALIDATOR` through the `DB_ENGINE` unit. The `crypt` example is interesting because it was involved in a documented vulnerability [9].

We implemented this design, which involved developing two decorator modules. The web server is running on our campus network, and we experimented with the design to ensure that boundary violations arising from improperly designed contracts are detected, even if the violation results from a call by a contract-unaware module. `PASSWD_VALIDATOR` and `DB_VALIDATOR` each use `CRYPT`, the former directly, the latter via `DB_ENGINE`. By the rules of our approach, only one process can negotiate a contract with the singleton instance of `CRYPT` at the same time. Thus, no two processes can access this unit simultaneously. Moreover, if a contract for either unit is omitted, access to the `CRYPT` unit results in an observable runtime exception instead of an unobserved corruption of the static variables of `crypt_module`. This property holds even when the `crypt` function is called from other contract-unaware code. For example, when the shared module underlying `DB_ENGINE` calls `crypt`, the call is redirected to the wrapper function, which checks whether the `CRYPT` unit is in the current realm. Absent a negotiated synchronization contract, this check fails and a runtime exception is raised. Furthermore, because the result of `crypt` is exported to thread-local memory, no two processes can share a reference to the result. Thus, aliasing errors are prevented as well.

6. Discussion and Related Work

We designed our model of synchronization contracts to protect against race conditions and aliasing errors, which manifest as serialization vulnerabilities in concurrent software systems. Our model protects reliably against all data races and protects against general races provided that the program is implemented in accordance with its contract. This paper contributes a technique for extending the protections that accompany programs developed entirely in a contract-aware language to contract-aware programs that use contract-unaware system or third-party modules.

Approaches for detecting serialization vulnerabilities in existing systems have been based on static analysis and testing. For example, static analysis is used to check if a program might exhibit interaction patterns that are known to produce vulnerabilities [6, 8, 24]. These approaches are limited to addressing vulnerabilities for which patterns are known and may produce false positives because of the conservative nature of static analysis. A testing approach focuses on systematic generation of test cases that are designed to detect race conditions [1]. Unfortunately, testing cannot guarantee that race conditions do not exist, and the number of test cases needed to exercise all interleavings of concurrent operations grows combinatorially. Such approaches are chiefly used within the “penetrate and patch”

¹⁰namely, a race condition involving the use of `crypt` [9] and an aliasing error where output was sent to the wrong connection [10].

paradigm, where an existing system is analyzed to find and remove security flaws [18]. While this paradigm may be necessary for legacy code, it is generally held to be inadequate for the engineering of secure systems from the ground up [22, 25]. In contrast, our approach seeks to minimize validation efforts: A client cannot access a supplier unless expressly permitted by a synchronization contract; any remaining serialization vulnerabilities result from a poorly-designed contract.

Most work on making C code secure focuses on preventing buffer overflows and memory management errors. Much like our approach, Cyclone employs ideas from type theory to ensure that only valid memory regions are accessed and that shared memory is accessed by a process only if it is locked [17, 16]. A dialect of C, it requires that code first be adapted to fit that dialect and then recompiled, but cannot generally be applied to unaltered C libraries in binary or source form.

Protecting C code against serialization vulnerabilities is necessary but not sufficient, because many other vulnerabilities, such as buffer overflows, are also prevalent in C libraries [11]. Therefore, our approach to securing C libraries will eventually have to be complemented with approaches such as CCured [21] or Safe C [2] that make C type-safe. Of particular interest are those approaches that facilitate working directly with binary libraries and do not require source code access or recompilation, such as SASI [14] or SELF [13]. To this end, we already use the conservative garbage collector by Boehm and Weiser [7] to protect against memory management errors.

In prior work, we evaluated the efficiency of contract negotiation on a Linux architecture and found it to be competitive with other approaches to synchronization [4]. In the absence of contention the cost of a realm update is linear in the number of units removed from or added to a realm, with a constant factor that is somewhat larger than the cost of the same number of primitive mutex operations would be. The higher constant results from the need to maintain book-keeping data structures in addition to the mutual-exclusion operations needed. In the presence of contention, the cost of realm updates is dominated by the cost for a process context switch, which occurs for any blocking algorithm.

Acknowledgements. Partial support for this research was provided by the Office of Naval Research grant N00014-01-1-0744 and by NSF grants EIA-0000433 and CCR-9901017.

References

- [1] Taimur Aslam. *A Taxonomy of Security Faults in the UNIX Operating System*. PhD thesis, Purdue University, 1995.
- [2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 290–301, 1994.
- [3] R. Behrends and R. E. K. Stirewalt. The universe model: An approach for improving the modularity and reliability of concurrent programs. In *Proc. of the 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-00)*, pages 20–29, 2000.
- [4] Reimer Behrends. *Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages*. PhD thesis, Michigan State University, December 2003.
- [5] A. Beugnard et al. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.
- [6] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.
- [7] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18:807–820, 1988.

- [8] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.
- [9] CVE Candidate. Can-2003-0189, 2003. URL:<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0189>.
- [10] CVE Candidate. Can-2003-0789, 2003. URL:<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0789>.
- [11] C. Cowan et al. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proc. of the DARPA Information Survivability Conference and Exposition*. IEEE Computer Society Press, January 2000.
- [12] W. Du and A. P. Mathur. Categorization of software errors that led to security breaches. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 392–407, 1998.
- [13] D. C. DuVarney, V. N. Venkatakrishnan, and S. Bhatkar. SELF: a transparent security extension for ELF binaries. In *Proc. of the 2003 workshop on new security paradigms*, pages 29–38, 2003.
- [14] Erlingsson and Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1995.
- [16] D. Grossman. Type-safe multithreading in cyclone. *ACM SIGPLAN Notices*, 38(3):13–25, March 2003.
- [17] D. Grossman et al. Region-based memory management in Cyclone. In *Proc. of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 282–293, Berlin, June 2002.
- [18] C. E. Landwehr et al. A taxonomy of computer program security flaws. *ACM Comput. Surv.*, 26(3):211–254, 1994.
- [19] B. Meyer. *Eiffel: the Language*. Prentice Hall, 1992.
- [20] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
- [21] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *Proc. of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, 2002.
- [22] C. Salter et al. Toward a secure system engineering methodology. In *Proc. of the 1998 workshop on New security paradigms*, pages 2–10. ACM Press, 1998.
- [23] R. E. K. Stirewalt, R. Behrends, and L. K. Dillon. A model-based semantics for synchronization contracts in object-oriented languages. Submitted for publication in the Elsevier journal *Science of Computer Programming*. An online copy may be found at <http://www.cse.msu.edu/~stire/Papers/scp.pdf>, 2004.
- [24] J. Viega et al. ITS4: a static vulnerability scanner for C and C++ code. In *Proc. of ACSAC 2000*, December 2000.
- [25] J. Viega and D. Evans. Separation of concerns for security. In *ICSE Workshop on Multidimensional Separation of Concerns in Software Engineering*, June 2000.