Challenges in defining a programming language for provably correct dynamic analyses

Eric Bodden, Andreas Follner*, and Siegfried Rasthofer**

Secure Software Engineering Group
European Center for Security and Privacy by Design (EC SPRIDE)
Technische Universität Darmstadt

Abstract. Modern software systems are not only famous for being ubiquitous and large scale but also infamous for being inherently insecure. We argue that a large part of this problem is due to the fact that current programming languages do not provide adequate built-in support for addressing security concerns.
In this work we outline the challenges involved in developing CODANA, a novel programming language for defining provably correct dynamic analyses. CODANA analyses form security monitors; they allow programmers to proactively protect their programs from security threats such as insecure information flows, buffer overflows and access-control violations. We plan to design CODANA in such a way that program analyses will be simple to write, read and prove correct, easy to maintain and reuse, efficient to compile, easy to parallelize, and maximally amenable to static optimizations. This is difficult as, nevertheless, CODANA must comprise sufficiently expressive language constructs to cover a large class of security-relevant dynamic analyses.
For deployed programs, we envision CODANA-based analyses to be the last line of defense against malicious attacks. It is hence paramount to provide correctness guarantees on CODANA-based analyses as well as the related program instrumentation and static optimizations.
A further challenge is effective but provably correct sharing: dynamic analyses can benefit from sharing information among another. We plan to encapsulate such shared information within CODANA program fragments.

Keywords: Runtime verification, inline reference monitors, code synthesis, declarative programming languages, information flow, buffer overflows

1 Introduction

Modern software systems are ubiquitous and often large scale, however many such systems are also inherently insecure. A large part of this problem is caused by the fact that currently programmers are forced to implement security features using general-purpose programming languages. While during the requirements

* At the time of writing, Andreas Follner was with the Technikum Wien.
** At the time of writing, Siegfried Rasthofer was with the Universität Passau.

elicitation phase of the software development process, software architects formulate security requirements rather concisely on a high level of abstraction, this simplicity becomes lost as appropriate security checks are implemented using generic low-level programming-language constructs.

As an example, consider the same-origin policy, an important security policy in web-based scripting languages such as JavaScript and ActionScript:

> "An origin is defined by the scheme, host, and port of a URL. Generally speaking, documents retrieved from distinct origins are isolated from each other." [35]

The same-origin policy can be concisely and precisely defined in a few paragraphs of English text. Implementing enforcement of the same-origin policy, however, is a whole different story, as is evident by a former violation of the same policy in WebKit [3], the rendering engine used in the Chrome [1] and Safari [2] browsers. Listing 1 shows change set 52401 in WebKit, which fixes a vulnerability that allowed for violations of the same-origin policy. The change comprises a single character; building WebKit involves downloading a software development kit of several gigabytes.[1]

This example shows the challenges involved with implementing security policies in large-scale software systems. Ideally, programming languages would allow for definitions of security policies at a high level and in a modular fashion, and implement the enforcement of those policies through automatic means. Today's reality, however, are low-level security checks in general-purpose languages, written and maintained by hand. The checks are scattered throughout the program, which makes them hard to trace and maintain. Moreover, they are tangled to the program's base functionality.

In this work we outline the challenges involved in developing CODANA, a novel programming language with which we try to rectify some of those problems. CODANA has the goal to be a language for defining provably correct dynamic analyses for security purposes. In this setting, dynamic analyses effectively form security monitors. Thus, they allow programmers to proactively protect their programs from security threats such as insecure information flows, buffer overflows and access-control violations. Opposed to design-time analyses, CODANA-based analyses are meant to remain a part of the program even after deployment; they form an essential security-critical part of the program.

[1] Building WebKit: `http://www.webkit.org/building/checkout.html`

```
1 - if(protocolIsJavaScript(url) ||
2 + if(!protocolIsJavaScript(url) ||
3       ScriptController::isSafeScript(newFrame) {
```

Listing 1: Fix for bug 30660 in WebKit (violation of same-origin policy)

CODANA is not a general-purpose programming language. Instead, we envision functional concerns of programs to be written in a "base language" such as Java or C/C++. CODANA-based analyses then uses aspect-oriented programming techniques to augment those base programs with instrumentation to fulfill the stated security goals.

At the time of writing, the language design for CODANA has not yet been fixed. In this paper we outline the challenges involved in designing such a language. We plan to design CODANA in such a way that program analyses will be simple to write, read and prove correct, easy to maintain and reuse, efficient to compile, easy to parallelize, and maximally amenable to static optimizations. On the other hand, CODANA must comprise sufficiently expressive language constructs to cover a large class of security-relevant dynamic analyses.

Dynamic analyses expressed in the CODANA language are not just supposed to be used to determine whether or not a program fulfills its security guarantees, but rather to implement security features that will establish those guarantees. A formerly insecure program hence becomes secure by augmenting it with dynamic analyses formulated in CODANA. This programming paradigm requires that dynamic analyses be efficient enough to actually remain part of the program even after deployment time. We hence plan to include a wide range of domain-specific static optimizations that restrict runtime checks to a necessary minimum.

In such deployed programs, CODANA-based analyses are likely to be the last line of defense against malicious attacks. It is hence paramount to provide correctness guarantees on CODANA-based analyses as well as the related program instrumentation and static optimizations.

A further challenge is effective but provably correct sharing and reuse: dynamic analyses can benefit from sharing information among another. We plan to encapsulate such shared information within reusable CODANA fragments. This fosters reuse of both CODANA implementations and correctness proofs.

To summarize, this paper provides the following original contributions:

- an outline of the challenges in designing a language for correct dynamic analyses,
- an outline of the impact of the language design on static optimizations to speed up those analyses,
- an outline of the requirements for providing correctness guarantees, and
- an outline of the potential for reuse of dynamic-analysis definitions.

The remainder of this paper is structured as follows. In Section 2, we discuss the trade-offs involved in CODANA's language design. Section 3 provides details about our envisioned static optimizations. Section 4 outlines the challenges involved in providing correctness proofs and guarantees. We discuss our plan to support sharing, reuse and extensions in Section 5. Section 6 discusses related work. We conclude in Section 7.

2 Dynamic Analysis

We next explain the challenges involved in designing a programming language for security-related dynamic analyses. First, one may ask why we opt at all to counter malicious attacks through dynamic and not static program analyses. The problem is that static-analysis tools are always limited in precision, as they have to make coarse-grain assumptions about the way a program is used, and which input a program is provided. In addition, all interesting static-analysis problems are inherently undecidable. In result, analysis result will always be approximate, which leaves static-analysis designers two options: design the analysis to be overly pessimistic or optimistic. An optimistic analysis would not be a viable option in a security-sensitive setting, as it would allow a potentially large class of malicious attacks to go unnoticed. A pessimistic static analysis, however, runs risk of generating false warnings. Such false warnings are a burden to the programmers, who are often under time pressure and have insufficient resources at their disposal to manually tell apart false warnings from actual vulnerabilities.

For those reasons, we base our approach primarily on dynamic runtime analysis. With a dynamic analysis, we can actually guarantee to detect certain classes of vulnerabilities without false warnings and without missed violations. For deployed programs, we envision CODANA-based analyses to be the last line of defense against malicious attacks. The analyses will identify vulnerabilities just in time, as they are about to be exploited. This allows the program to induce countermeasures to prevent the exploit from succeeding.

We would like CODANA-based analyses to be able to detect and mitigate different kinds of attacks, such as attacks based on buffer overflows, insecure information flows and cross-site scripting, circumvention of access control, exploitation of leaked capabilities, and side channels such as timing channels. To this end, CODANA needs to support various language features. To identify buffer-overflows, one must be able to reason about numeric values and operations, as well as pointer assignments. Insecure information flows and cross-site scripting vulnerabilities can only be identified if the sources of sensitive information are known and if values assigned from those sources can be tracked trough all possible program operations. Access-control and object-capabilities require an analysis to be able to associate state with objects. Timing channels require an analysis to reason about real-time data.

In the following, we explain some of those requirements in more detail by given two examples: the detection of buffer overflows and a mechanism for enforcing access control. The reliable detection of buffer overflows during runtime could be realized by comparing the lengths of the buffers right before a vulnerable function like `strcpy` is called.

Listing 2 shows what language constructs in CODANA could look like that could support such a use case. We here use a syntax roughly based on a related static-analysis approach by Le and Soffa [28]. Anytime the `strcpy` function is called, the CODANA program compares the lengths of the two parameters and, in case the length of the source buffer exceeds the length of the destination buffer, raises a violation. To support the user with a concise syntax, the language

```
1 Buffer a,b;
2 at 'strcpy(a,b)' if len(a) < len(b) violation(a)
3 violation(Buffer a) {
4   print("buffer overflow detected in variable " +
5   name(a) + " at " + location); }
```

Listing 2: Detecting buffer overflows with CODANA (based on [28])

will provide built-in constructs such as `len`, which represents the length of a selected buffer, and `location`, which represents the current code location. Most of those constructs will require runtime support. For instance, to be able to tell the length of a buffer, the CODANA runtime must track this value in the first place. We plan to provide the necessary program instrumentation through technologies from aspect-oriented programming [27]. The difference between CODANA and general-purpose aspect-oriented programming languages is that CODANA requires a more fine-grained approach. For instance, languages like AspectJ [8] allow users to instrument calls to methods and assignments to fields but not assignments between local variables. In this respect, CODANA can be seen as a domain-specific aspect language, for the domain of security monitoring.

As another example of a use case that we envision the CODANA language to support, consider the problem of access control. To this end, we plan to have CODANA support specially associative arrays[2] that can be used to keep track of a user's authorizations.

Listing 3 shows how one could use an `enum` construct and associative arrays to model a dynamic analysis detecting access violations. In the security community, such dynamic analyses are frequently called security automata [33] or inline reference monitors [23]. Lines 1–2 define two different classes of internal states that we use to keep track of whether a user is currently logged in and whether or not the user has been granted access to a given file. Note that we include such constructs for modeling finite states on purpose. We plan to conduct effective, domains-specific optimizations to CODANA programs (see Section 3), and those are easier to conduct when data structures are known to be finite. In lines 4–5, we use two associative arrays to map users and files to their respective states. Note that often one will encounter situations in which states must be associated with combinations of objects such as in line 5, where we associate a state with a user and file. Line 7 defines local variables `u` and `f`. The remainder of the code uses those typed variables as place holders for runtime objects. Lines 9–12 define four rules (or pieces of advice) to update the security monitor's state based on a range of concrete program events. Lines 9–12 define an error handler. Whenever the underlying program calls the method `fgets`, we check whether the third

[2] An associative array is an array that can be indexed not just by numbers but by objects. Although associative array appears syntactically just as normal arrays, they are typically implemented through map data structures.

```
1  enum LoginState { LOGGED_OUT, LOGGED_IN }
2  enum Access { GRANTED, FORBIDDEN }
3
4  LoginState[User] loginState = LOGGED_OUT;
5  Access[User,File] access = FORBIDDEN;
6
7  User u, File f;
8
9  after 'u=login()' loginState[u] = LOGGED_IN;
10 after 'logout(u)' loginState[u] = LOGGED_OUT;
11 after 'grantAccess(u,f)' access[u,f] = GRANTED;
12 after 'revokeAccess(u,f)' access[u,f] = FORBIDDEN;
13
14 at 'fgets(*,*,f)' with 'u=curr_user()'
15   if loginState[u] != LOGGED_IN ||
16      access[u,f] != GRANTED violation(u,f);
```

Listing 3: Access control with CODANA

argument, the file `f`, may be accessed by user `u`, who is fetched from the current context.

Expressiveness vs. simplicity We plan to design CODANA in such a way that it is not only simple to use, but also is amenable to correctness proofs and static optimizations. Efficiency is a big concern for CODANA. If no due care is taken, dynamic analysis can slow down a program's execution considerably [14,19]. This calls for a language design that focuses on simplicity. The simpler the language constructs that CODANA supports the easier it will be, both for compilers and for programmers, to prove properties about CODANA-based analyses. Frequently found features in general-purpose programming languages that cause problems for static analyses are infinite state, pointers and aliasing, loops and recursion as well as exceptions. While it may be necessary for CODANA to comprise some of those features, we plan to thoroughly investigate, which features to include, and how to make programmers aware of the performance or maintenance penalties that their use may entail.

Use of infinite state could be excluded or at least discouraged by supporting language constructs like `enum`, which we mentioned above. Aliasing could be excluded by adapting a pass-by-value semantics for variables. In general, this may increase analysis runtime, as every assignment entails a deep copy. However, static optimizations could counter this effect. Loops could at least be restricted to bounded *for-each*-style loops. Recursion at this point seems unnecessary to include in CODANA altogether.

Another important matter is concurrency. On the one hand, we wish to include constructs that enable CODANA to detect data races [16,17]. On the other hand, our own data structures need to be thread safe, and preferably, for per-

formance reasons, lock-free as well. We plan to design and implement such data structures in the back-end of CODANA, e.g. to implement runtime support for associative arrays.

3 Static Optimization

We envision CODANA to be used to secure end-user programs that are deployed at the user's site. But dynamic program analysis often requires an extensive amount of program instrumentation, which can slow down the analyzed program considerably [14, 19]. The fact that CODANA will support the analysis of data-centric information flows (information-flow analysis) such as insecure information flows or access-control violations yields CODANA programs that have to track a considerable amount of runtime information. Much of the overhead is attributable to the fact that each variable could track different data-centric or security-centric information. To improve the dynamic analysis, we and others have shown in the past that a static analysis can be very effective in speeding up dynamic analyses [15–17, 19, 22, 36]. These approaches, also frequently called hybrid program analyses, usually build on the idea of only instrumenting certain program parts, while at the same time proving that instrumentation of other parts of the program is unnecessary: monitoring those program parts would have no effect on the outcome of the dynamic analysis. Those parts are identified in advance, through static analysis of the program to be monitored with respect to the definition of the dynamic analysis. The static analysis is used to eliminate useless instrumentations which causes a reduction of events dispatched to the dynamic-analysis code, hence reducing its evaluation time. In the past, we have also applied proof techniques to formally show that our static optimizations are correct, i.e., that they do not change the outcome of the dynamic analyses [14, 15]. So far, this approach is based on control-flow analysis, but we plan to extend the approach to information-flow analysis as well.

Let's consider a simple data-centric policy rule which is efficiently enforced by a typestate analysis as described in [14]. The data-centric policy is a modified version of the secure coding guideline *Sanitize the Output* taken from Aderhold et. al [4]. Figure 1 shows the simplified taint-flow finite-state machine which could be used as a runtime monitor for the detection of Cross-Side-Scripting attacks. In CODANA, such state machines could be expressed via enums, such as shown in Listing 3.

This finite-state machine contains three different states whereas s_0 and s_1 are security-irrelevant states, whereas the *error* state symbols a policy violation (Cross-Side-Scripting attack). There are also three different kind of events (*tainted*, *untainted* and *output*) which get activated by different program statements. For example, the event *tainted* gets activated by `$_GET['tainted_data']`, the *untainted* event by statements which assign definitely untainted values and the *output* event is activated if the data leaks from the program, for instance when data is printed to the browser.

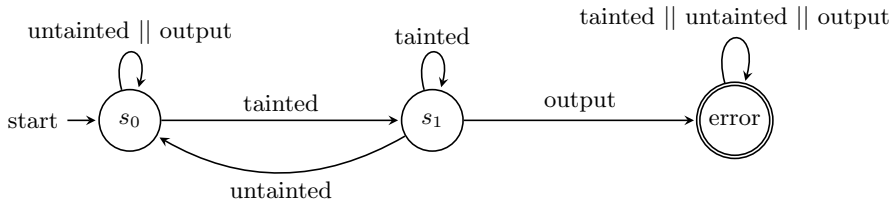

Fig. 1: Simple taint-flow finite-state machine for the prevention of Cross-Side-Scripting attacks

An information-flow analysis would associate such a state machine with each tracked variable. Each variable starts in the initial state (s_0) and performs a transition corresponding to the activated event. Listing 4 shows an example with tainted and untainted data and also one security-relevant flow along line 1 → line 5 → line 7, which could allow a Cross-Side-Scripting attack. With the tracking of the different security events and the corresponding transitions in the finite-state machine, the analysis is able to identify this kind of attack if one of the variables reaches an error state.

A general, un-optimized dynamical-analysis approach would instrument each assignment, as shown in 4. In this example, however, the instrumentation of the untainted variable *$number* is completely unnecessary: on this variable, no taint violations can take place, and hence the analysis would never report taint violations on this variable. A static information-flow analysis, executed in advance, would allow the CODANA compiler to omit instrumentation for this variable from the dynamic analysis. The result after applying the static analysis and optimization is shown in Listing 5.

```
1  $input1 = $_GET['tainted_data'];
2    makeTransition(input1, tainted);
3  $number = 1;
4    makeTransition(number,
         untainted);
5  $input2 = $input1;
6    propagateTaint(input1, input2);
7  echo($input2);
8    makeTransition(input2, output);
9  echo($number);
10   makeTransition(number, output);
```

Listing 4: Example exposing a Cross-Side-Scripting attack without static optimization

```
1  $input1 = $_GET['tainted_data'];
2    makeTransition(input1, tainted);
3  $number = 1;
4  $input2 = $input1;
5    propagateTaint(input1, input2);
6  echo($input2);
7    makeTransition(input2, output);
8  echo($number);
```

Listing 5: Example exposing a Cross-Side-Scripting attack with static optimization

A significant challenge to such static optimizations are multi-threaded programs. For such programs, multiple control-flows can be interleaved. In consequence, a single control-flow graph is not sufficient to simulate all possible control flows. Moreover, the analysis state quickly grows due to the many possible different schedules that static analyses need to simulate. Many existing whole-program analysis (including some of our own previous work [14]) ignore

this problem. A promising escape route are flow-insensitive analyses [19]. Such analyses do not at all take the program's control-flow into account. Because of this, the analyses are, by design, agnostic to the different possible schedules. At the same time, such analyses can be implemented quite efficiently.

We hence plan to follow a staged analysis approach that applies relatively inexpensive flow-insensitive analysis first. As we observed in previous work [19], such analyses can often optimize away already a significant amount of program optimization. We then execute more expensive, potentially thread-aware, flow-sensitive analyses only to such parts of the program in which instrumentation remains after the first analysis stages have been applied.

But multi-threading is not just an annoyance but can also be of help. We plan to investigate to what extent our static-analysis algorithms can be designed to exploit parallelism. Rodriguez and Lhoták have recently shown [31] that such an approach promises significant speed-ups. In addition, parts of the CODANA runtime could be designed to support executing the dynamic analysis in separate threads.

4 Correctness

Dynamic analyses based on CODANA will usually be able to detect bugs and vulnerabilities just as they are about to be exploited. Because of this, the analyses are practically the program's last line of defense. It is hence paramount that analyses written in CODANA be reliable. We plan to prove the correctness of CODANA programs on several levels.

One threat to the correctness of CODANA-based analyses are the static optimizations that we apply. In previous work we have demonstrated how a proof technique based on so-called continuation-equivalent analysis configurations can be used to prove the correctness of such optimizations [15]. In a nutshell, one must prove that if a static optimization removes instrumentation at a statement s, then all possible analysis configurations before and after s must be equivalent with respect to all possible continuations of the control flow that follow s. If they are equivalent, then this means that dynamically executing the instrumentation at s would have no effect, and hence it is sound to omit the instrumentation at this statement. In the past, we have used this approach to prove the correctness of a flow-sensitive static typestate-analysis [13, 14]. This process also revealed bugs in previous approaches [20, 30]. For CODANA, we plan to extend this approach to other classes of static optimizations for dynamic analyses.

CODANA programs consist mainly of program instrumentation and accesses to a runtime library, both of which need to adhere to correctness guarantees. In recent work, we have developed a clean semantics for weaving of aspect-oriented code into Java programs [25]. We assume to be able to reuse some of the results to prove that our instrumentation preserves the behavior of the instrumented program. A challenge in this area are race conditions and side-channel attacks. As the instrumentation caused by our dynamic analysis causes the program to slow down, this may cause certain race conditions or certain

information leaks, e.g., through timing channels to disappear due to this slowdown. Such so-called "Heisenbugs" are a general problem in dynamic analysis that cannot be solved without specific modifications to the program's scheduler. Essential parts of CODANA's runtime library could be proven correct through tool-assisted functional-correctness proofs [37].

We plan to aid programmers in proving the correctness of analyses formulated in CODANA. Given a high-level security property, programmers should be able to argue why a given CODANA program establishes this property. To this end, we first plan to keep the language itself as simple as possible (see Section 2), but also plan to include a standard library with CODANA code templates. Along with those templates, we can provide example proofs that prove important properties about those templates. Ideally, those proofs could then be composed to a correctness proof for a CODANA program that uses the respective code templates.

5 Reuse, Sharing & Composition

In the previous section, we have already explained the advantages of a standard library for CODANA programs. In addition to this kind of reuse, we still plan to support reuse on other levels.

For instance, a common use case will be that programs execute augmented not with only one single dynamic analysis but with multiple ones. For instance, one may want to secure a program against information-flow violations and buffer overflows at the same time. Both of those information need to track assignments to certain classes of variables. When both analyses are performed at the same time, it is hence advisable to share information among those analyses. This sharing must be correct, however, it must not lead to unintentional alterations of the analysis information.

There are multiple ways to implement such information sharing. A simple way would be to provide certain common analysis elements as parts of the CODANA runtime library. If multiple analyses include the same elements and are executed at the same time, then this could lead to automatic sharing. A drawback of this approach is that we as CODANA designers must be able to anticipate common use cases for sharing to provide them in such a library. Another, more sophisticated approach, could try to identify the potential for information sharing irrespective of the origin of the analysis code. Such an approach would require a sophisticated analysis of the CODANA programs. In recent work, we have outlined the challenges that arise from composing instrumentations for multiple dynamic analyses [7].

Many of our static analyses and optimizations, although domain specific, may have parts that are reusable also for other static-analysis problems. We plan to encapsulate those analyses such that they can be reused by others. In the past, we have made accessible static analyses through simple domain-specific extensions to AspectJ [16–18,21]. A similar approach could be taken also in this project.

In addition, we plan to open our compiler implementation up to others. That way, other researchers could extend CODANA with additional language constructs or different static optimizations, such as we and others have previously done with AspectJ [10]. In the past, we have developed the Clara framework, which is explicitly designed to allow analysis extensions by others [18, 21].

6 Related Work

One of the most closely related projects is ConSpec [6], another formal specification language for security policies. As we propose for CODANA, also ConSpec supports advice-like before/after blocks that allow users to update a finite set of state variables. ConSpec allows for the definition of two different entities, called *policies* and *contracts*, both of which are defined manually by the user and are written in the ConSpec language. Contracts are application specific and describe the kinds of security properties that an application guarantees. Contracts can be checked against applications through a translation into Spec# [11] and subsequent static verification [5]. Policies are more general than contracts. They are specific with respect to an execution environment, e.g., a device on which the program is to be executed. ConSpec assumes that both policies and contracts are finite-state, which allows ConSpec to use simple algorithms for deciding regular-language inclusion to decide whether a contract complies with a policy. Further, ConSpec allows the monitoring of policies against applications, either through an external monitor or through an inline reference monitor [23]. We believe that the distinction between policy and contract is an interesting and valuable one. Similar concepts may be useful also for CODANA. On the other hand, CODANA will go much beyond what is supported by ConSpec, in that it will allow the generation runtime monitors that are statically optimized, and nevertheless will provide language constructs like associative arrays, which go beyond finite state. In previous work, we have developed Join Point Interfaces [24, 25], a mechanism to establish clean interfaces for aspect-oriented programs. Those interfaces currently focus on establishing the ability to type-check aspects independent of the base program's code. It may be useful to combine mechanisms of those join point interfaces with some of those of ConSpec within CODANA to achieve a separation between policies and contracts.

Le and Soffa present a generative approach that has some similarity to CODANA [28]. The approach provides a domain-specific specification language for program analyses. In the case of Le and Soffa, however, this approach is restricted to purely static analyses. Programmers can use the language to define how static-analysis information needs to be updated at particular classes of statements, and which conditions on the analysis information signal property violations. Based on the specification, the approach then automatically generates an appropriate flow-sensitive and path-sensitive static analysis for C/C++ programs. The authors demonstrate the efficacy of their approach by implementing analyses to detect buffer overflows, integer violations, null-pointer de-references and memory leaks. Our approach will provide a language that may have simi-

larities with what Le and Soffa propose. However, due to the fact that we focus on dynamic analysis, we may be able to provide certain language features that static analyses cannot provide, and vice versa. Moreover, we plan to not focus on C/C++ programs but rather on an intermediate representation that allows us to instrument and analyze programs written in a range of different languages.

DiSL, a domain-specific language for bytecode instrumentation by Marek et al., is another very related project [29]. DiSL is currently implemented not as a programming language with own, domain-specific syntax, but rather as a set of annotations and conventions over syntactic constructs defined in pure Java. Using DiSL, programmers can define pieces of advice to be applied before or after certain sequences of Java bytecode. DiSL further provides convenience methods for accessing elements on the stack or from other parts of the execution context. As DiSL programs are compiled, accesses to those methods are then automatically replaced by low-level (stack) operations. One important advantage of DiSL over other instrumentation tools is that DiSL allows for the uniform instrumentation of *entire* Java programs, including relevant parts of the Java runtime library. Codana differs from DiSL in that it will provide domain-specific programming constructs with a simple and well-defined semantics. The intricacies of bytecode instrumentation will be hidden from the user. This not only suggests that Codana programs may be easier to read and understand that programs written in DiSL, but also that they are more amenable to static optimizations. It may be interesting, though, for Codana to use DiSL as a back-end instrumentation technology, and we are currently discussing this opportunity with the developers of DiSL.

In the past, the first author has developed the Clara [18, 21] framework for static typestate analysis. Similar to the approach we propose here, also Clara uses static optimizations to speed up dynamic analyses. Also Clara provides a domain-specific aspect language for this purpose. In contrast to Codana, however, Clara is restricted to finite-state runtime monitors, and hence only supports static typestate analyses. While Codana will reuse some ideas of Clara, in this paper we showed that implementing a language such as Codana comes with many challenges that go beyond our previous experience with Clara.

Austin and Flanagan present a purely dynamic information-flow analysis for JavaScript. Their approach "detects problems with implicit paths via a dynamic check that avoids the need for an approximate static analyses while still guaranteeing non-interference" [9]. We plan to investigate whether we can use similar tricks in our implementation of Codana. Zhivich et al. compare seven different dynamic-analysis tools for buffer overflows [38]. XSS-Guard [12] by Bisht and Venkatakrishnan is a dynamic approach for detecting cross-site scripting attacks. The approached is based on a learning strategy; it learns the set of scripts that a web application can create for any given HTML request. This is different from Codana in that it gathers information among multiple program runs. We will investigate whether such an extension of the scope of Codana can be of more general use. Vogt et al. [34] implement a hybrid dynamic/static analysis to find cross-site scripting vulnerabilities. Interestingly, they use static analysis not to

enhance efficiency, but to detect attacks that through a purely dynamic analysis may go unnoticed. We plan to investigate whether such analyses would be useful to have within CODANA.

Jones and Kelly propose an approach to dynamically enforce array bounds through the use of a table which holds information about all valid storage elements [26]. The table is used to map a pointer to a descriptor of the object to which it points, which contains its base and extent. To determine whether an address computed off an in-bounds pointer is in bounds, the checker locates the referent object by comparing the pointer with the base and size information stored in the table. Then it checks if the new address falls within the extent of the referent object. The authors implemented their bounds checking scheme in the GNU C compiler (GCC), where it intercepts all object creation, address manipulation and de-reference operations and replaces them with their own routines. A problem observed with their approach is that it sometimes incorrectly crashes working code and that it considerably slows down program execution. Ruwase and Lam took the basic concepts, improved them and created CRED (C Range Error Detector) [32], which eradicated mentioned problems. We will investigate if some of the basic ideas used in either of the approaches could be adapted for CODANA.

7 Conclusion

We have presented a range of important design decisions involving the development of CODANA, a novel programming language for correct dynamic analysis. Challenges arise in the areas of dynamic analysis, static optimization, correctness, as well as reuse, information sharing and analysis composition. CODANA has the goal to allow programmers to write dynamic program analyses that will be simple to write, read and prove correct, easy to maintain and reuse, efficient to compile, easy to parallelize, and maximally amenable to static optimizations. We have explained how we wish to achieve those goals, and which implications those goals will probably have on the language design.

Acknowledgements. This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED. We thank Andreas Sewe, Walter Binder and Mira Mezini for discussions and suggestions on the topics presented in this paper.

References

1. Chrome Browser. `https://www.google.com/chrome`.
2. Safari Browser. `http://www.apple.com/safari/`.
3. The WebKit Open-Source Project. `http://www.webkit.org/`.
4. M. Aderhold, J. Cuéllar, H. Mantel, and H. Sudbrock. Exemplary formalization of secure coding guidelines. Technical Report TUD-CS-2010-0060, TU Darmstadt, Germany, 2010.

5. I. Aktug, D. Gurov, F. Piessens, F. Seehusen, D. Vanoverberghe, and E. Vétillard. Static analysis algorithms and tools for code-contract compliance, 2006. Public Deliverable D3.1.2, S3MS, `http://s3ms.org`.
6. I. Aktug and K. Naliuka. ConSpec–a formal language for policy specification. *Electronic Notes in Theoretical Computer Science*, 197(1):45–58, 2008.
7. D. Ansaloni, W. Binder, C. Bockisch, E. Bodden, K. Hatun, L. Marek, Z. Qi, A. Sarimbekov, A. Sewe, P. Tuma, and Y. Zheng. Challenges for refinement and composition of instrumentations (position paper). In *International Conference on Software Composition (SC 2012)*, May 2012. To appear.
8. The AspectJ home page, 2003.
9. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.
10. P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 87–98, New York, NY, USA, 2005. ACM.
11. M. Barnett, K. Leino, and W. Schulte. The spec# programming system: An overview. *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69, 2005.
12. P. Bisht and V. Venkatakrishnan. Xss-guard: precise dynamic prevention of cross-site scripting attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, 2008.
13. E. Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, June 2009. Available in print through ProQuest.
14. E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 5–14, New York, NY, USA, 2010. ACM.
15. E. Bodden. Continuation equivalence: a correctness criterion for static optimizations of dynamic analyses. In *WODA '11: International Workshop on Dynamic Analysis*, pages 24–28. ACM, July 2011.
16. E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA 2008), Seattle, WA*, pages 155–165, New York, NY, USA, July 2008. ACM.
17. E. Bodden and K. Havelund. Aspect-oriented race detection in Java. *IEEE Transactions on Software Engineering (TSE)*, 36(4):509–527, July 2010.
18. E. Bodden and L. Hendren. The Clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–20, 2010.
19. E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, pages 525–549, July 2007.
20. E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16)*, pages 36–47, New York, NY, USA, 2008. ACM.
21. E. Bodden, P. Lam, and L. Hendren. Clara: a framework for statically evaluating finite-state runtime monitors. In *1st International Conference on Runtime Verification (RV)*, volume 6418 of *LNCS*, pages 74–88. Springer, Nov. 2010.

22. M. B. Dwyer and R. Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 124–133, New York, NY, USA, 2007. ACM.
23. U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, 2003.
24. M. Inostroza, Éric Tanter, and E. Bodden. Modular reasoning with join point interfaces. Technical Report TUD-CS-2011-0272, CASED, Oct. 2011.
25. M. Inostroza, E. Tanter, and E. Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *ESEC/FSE '11: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 508–511, 2011.
26. R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, pages 13–26, 1997.
27. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP'97Object-Oriented Programming*, pages 220–242, 1997.
28. W. Le and M. L. Soffa. Generating analyses for detecting faults in path segments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 320–330, New York, NY, USA, 2011. ACM.
29. L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. Disl: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 239–250, New York, NY, USA, 2012. ACM.
30. N. A. Naeem and O. Lhotak. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 347–366, New York, NY, USA, 2008. ACM.
31. J. Rodriguez and O. Lhoták. Actor-based parallel dataflow analysis. In J. Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 179–197. Springer Berlin / Heidelberg, 2011.
32. O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.
33. F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.
34. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42, 2007.
35. W3C. Same-Origin Policy. `http://www.w3.org/Security/wiki/Same_Origin_Policy`.
36. S. H. Yong and S. Horwitz. Using static analysis to reduce dynamic analysis overhead. *Form. Methods Syst. Des.*, 27(3):313–334, Nov. 2005.
37. K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 349–361, New York, NY, USA, 2008. ACM.
38. M. Zhivich, T. Leek, and R. Lippmann. Dynamic buffer overflow detection. In *Workshop on the evaluation of software defect detection tools*, volume 2005, 2005.