Cinnamon: A Domain-Specific Language for Binary Profiling and Monitoring

Mahwish Arif
University of Cambridge
Cambridge, UK
mahwish.arif@cl.cam.ac.uk

Ruoyu Zhou
University of Cambridge
Cambridge, UK
zhoubot@gmail.com

Hsi-Ming Ho
University of Sussex
Brighton, UK
Hsi-Ming.Ho@sussex.ac.uk

Timothy M. Jones
University of Cambridge
Cambridge, UK
timothy.jones@cl.cam.ac.uk

2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 978-1-7281-8613-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CGO51591.2021.9370313

***Abstract*—Binary instrumentation and rewriting frameworks provide a powerful way of implementing custom analysis and transformation techniques for applications ranging from performance profiling to security monitoring. However, using these frameworks to write even simple analyses and transformations is non-trivial. Developers often need to write framework-specific boilerplate code and work with low-level and complex programming details. This not only results in hundreds (or thousands) of lines of code, but also leaves significant room for error.**

To address this, we introduce Cinnamon, a domain-specific language designed to write programs for binary profiling and monitoring. Cinnamon's abstractions allow the programmer to focus on implementing their technique in a platform-independent way, without worrying about complex lower-level details. Programmers can use these abstractions to perform analysis and instrumentation at different locations and granularity levels in the binary. The flexibility of Cinnamon also enables its programs to be mapped to static, dynamic or hybrid analysis and instrumentation approaches. As a proof of concept, we target Cinnamon to three different binary frameworks by implementing a custom Cinnamon to C/C++ compiler and integrating the generated code within these frameworks. We further demonstrate the ability of Cinnamon to express a range of profiling and monitoring tools through different use-cases.

***Index Terms*—Domain-Specific language, Profiling, Binary analysis and instrumentation**

I. Introduction

Profiling and monitoring tools that work directly with application binaries are valuable aids for understanding and characterizing program execution, as well as for detecting bugs and ensuring they run safely. They are especially important when application source code is unavailable, recompilation is infeasible, or it uses third-party applications and external libraries [1]. However, writing these tools from scratch is a challenging and tedious task and generally involves a process of disassembly or lifting the binary to a higher-level representation that is easier to work with. Programmers then also need to write code to recover control-flow information, such as basic blocks and functions, which adds to the development challenge [2].

To address this, a range of tools have been developed that can lift a binary to assembly [3], [4], LLVM's intermediate representation (IR) [5] or a custom IR [6]–[8]. There are also a number binary analysis and instrumentation frameworks [9]–[15], that provide infrastructure support to write custom profiling and monitoring tools. However, developing such tools is a non-trivial task, even for simple profiling and monitoring schemes, due to the complexity of these frameworks and different programming interfaces. Developers not only need to add boilerplate code to implement their algorithms within these frameworks, but may also need to write complex low-level code to perform even simple accesses or modifications to the binary, such as finding the address loaded from or stored to by a memory instruction [9]. This adds significantly to the length and complexity of the code and increases the chance of bugs creeping into the final tool.

On the other hand, were these tools to be more accessible to developers then they would be able to perform sophisticated analysis and profiling of all manner of applications. Providing developers with a simple interface to the underlying frameworks would avoid them having to write low-level and repetitive code, allowing them to focus solely on the correctness of their analysis and monitoring techniques. This would unleash a new era of binary instrumentation, with renewed interest in developing schemes that operate at the binary level, independent of any particular compiler framework, which is becoming increasingly important as more code gets dynamically generated by a JIT and developers have a wider choice of source languages to write in.

In order to overcome these challenges, we propose a domain-specific language (DSL), Cinnamon, to write programs for binary profiling and monitoring. Cinnamon raises the abstraction level for developers, providing a natural programming model in which to write analysis and instrumentation code more efficiently and in a framework- and platform-independent way. The major contributions of this work include:

- the design and implementation of Cinnamon, a DSL and compiler tool-chain for writing binary profiling and monitoring programs;
- higher-level control-flow, data and type abstractions to facilitate efficient implementation of different profiling and monitoring techniques;
- a programming model and abstractions to facilitate mapping of Cinnamon programs to static, dynamic or hybrid binary instrumentation frameworks;

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through grant reference EP/P020011/1. Additional data related to this publication is available in the repository at https://doi.org/10.17863/CAM.62760.

- a set of case studies demonstrating the applicability and flexibility of Cinnamon to express different binary profiling and monitoring techniques from the literature;
- an open-source implementation of Cinnamon[1].

II. Background

The ability to perform analysis and instrumentation directly on program binaries has many powerful applications, from profiling to security monitoring. Code-coverage profilers [14], [16] and performance analyzers [17] help in identifying program hotspots with the potential for optimization. Security monitoring tools can analyze a binary to find malware signatures [18], [19] or add instrumentation to monitor its execution, so as to detect or prevent security attacks [20], [21]. These can also be used to identify security vulnerabilities [9], [22]–[25] that can be exploited by malicious attackers. Beyond passive profiling and monitoring, other techniques actively transform the binary to optimize performance through the code that is executed [26]–[28] or its layout in memory [29].

Frameworks for binary instrumentation that facilitate these applications can operate on the binary statically, dynamically, or a combination of the two. Each has its pros and cons that we now explore.

A. Static Binary Frameworks

Static binary frameworks enable binary analysis, instrumentation or rewriting without executing the application, avoiding the run-time cost of performing the analysis or code modifications [15]. Due to this, complex and time-consuming analyses can be performed, such as those that explore all possible execution paths through the program, and be more complete with higher application code coverage. Recovery of higher-level structural information also becomes more feasible.

A number of frameworks support the development of custom binary tools using static approaches [7], [14], [15], [30], [31] for complex control-flow, data-flow and dependency analyses [28], [32] and for optimization and security [33]–[35]. Related frameworks support link-time binary rewriting, such as Diablo [36] and Alto [37].

However, static binary frameworks can suffer from difficulties in recovering an accurate control-flow graph in the presence of indirect jumps, which may hamper accurate global analysis. They are also conservative in their analysis and may raise false positives [38] due to the lack of precise run-time information. Furthermore, binaries with dynamic linkage cannot be analyzed beyond the shared-library call, a similar problem to that faced by compilers. Finally, it is challenging to verify that the rewrites made by these static tools are correct.

B. Dynamic Binary Frameworks

Dynamic binary analysis and transformation techniques can be useful where execution depends on the run-time environment or critical information is not known statically. For example, execution may actually only proceed down a small subset of all control-flow paths and indirect branches. Hence, dynamic schemes can limit themselves to the code paths actually seen during execution, in contrast to static techniques. Dynamic frameworks are also critical for analyzing programs with interrupts and calls to shared libraries, which are not present in the static binary, and detecting dynamically generated or self-modifying malicious code.

Dynamic binary frameworks enable analyses and transformations to be applied while the code is executing. Valgrind [9] is a popular dynamic binary instrumentation framework that has facilitated the development of a number of profiling [39], [40] and vulnerability detection tools [41]. Valgrind disassembles the binary to a custom IR called VEX and resynthesizes to machine code after adding instrumentation. Pin [10], DynamoRIO [11], and StarDBT [42] use dynamic binary translation to add analysis code or perform transformations. Tools built using these frameworks carry out a variety of functions, such as performance analysis [43], memory debugging [22], taint checking [44], [45] and protection against control-flow attacks [21].

Dynamic analysis can be more precise than static approaches but lacks a global overview of the application and has lower code coverage. Writing dynamic analysis tools is generally a complex, tedious and error-prone process and may involve inserting trampoline calls in the instruction stream to developer-written analysis routines and ensuring that context is saved and restored properly. Moreover, the high run-time overhead of dynamic techniques hinders the development of complex schemes and limits the usability of these techniques to relatively simple analyses and transformations [9].

C. Static-Dynamic Hybrid Binary Frameworks

Both static and dynamic approaches have their strengths and weaknesses. However, these approaches are complementary to each other and can be combined to build more powerful tools that can reduce the limitations of each approach when they are applied separately.

BitBlaze [12] is a binary analysis platform specifically built for security applications and features components for static analysis, dynamic analysis and symbolic execution. Another tool, angr [38], uses static binary analysis and dynamic symbolic execution to perform different analyses, such as control-flow recovery and vulnerability detection.

Dyninst [13] offers both static and dynamic binary rewriting approaches and has been used to build tools for profiling and performance analysis [46]. Janus [26] is a hybrid static-dynamic binary modification framework based on DynamoRIO that combines the power of both static and dynamic approaches to perform complex analyses and transformations, such as binary parallelization and vectorization.

Although these frameworks build on the strengths of static analysis with dynamic rewriting, they also inherit their weaknesses too. Developers have to deal with low-level operations when writing both the static passes that analyze the binary and the dynamic tools that perform modification. As such, although they are more powerful frameworks, they can be more difficult to use. What is needed is a method of programming

[1]Cinnamon is available at https://github.com/CompArchCam/Cinnamon.

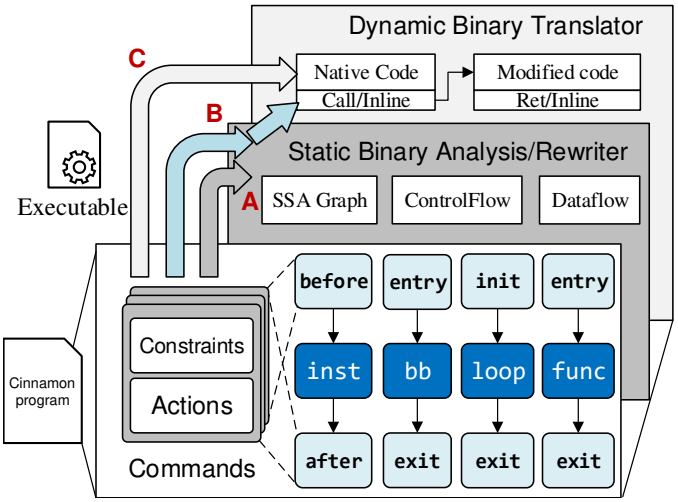

Figure 1: Overview of a program written in Cinnamon.

```
1  <global variable declarations>
2  /*-------- command 1 starts --------*/
3  inst I <constraint> {
4    <local variable declarations>
5    /* inspection or analysis code */
6    <statements>
7    /* action 1 */
8    before I {
9      <local variable declarations>
10     <statements>
11   }
12   /* action 2 */
13   after I {
14     <statements>
15   }
16 }
17 /*-------- command 1 ends --------*/
```

Figure 2: High-level pseudo-code of a Cinnamon program for instrumenting instructions.

binary instrumentation frameworks of all flavors at a higher level of abstraction, so as to avoid having to deal with low-level details. The next section presents Cinnamon, our domain-specific language and compiler to provide this abstraction to ease the burden of writing instrumentation code.

III. The Cinnamon Language

We designed Cinnamon for building binary analysis and instrumentation tools for static, dynamic, and hybrid frameworks. Cinnamon abstracts away details of the target binary analysis and instrumentation framework and processor ISA, allowing programmers to focus on the analysis that they wish to perform, rather than the details of how to go about doing it. Many programs written in Cinnamon are portable across target frameworks, regardless of the type of instrumentation performed or the architecture they target. The Cinnamon compiler deals with the complexities of mapping the high-level instrumentation intent, written in Cinnamon, to the specific implementation required for binary instrumentation within a given framework.

A. Overview

Figure 1 shows a diagrammatic overview of the core components of a program written in Cinnamon and its mapping to different binary frameworks.

A Cinnamon program is made up of a series of *command* blocks, each of which operates on a logical control-flow element (CFE) from the binary, such as an instruction, loop or function. A command specifies some analysis and *actions* to be instrumented at different execution points of a CFE based on certain *constraints*.

A command in Cinnamon is mapped by the compiler to a target backend instrumentation framework. For a purely static or dynamic framework, a command is mapped in its entirety to a static or dynamic tool for that framework respectively. If the target framework is a hybrid design then the analysis and instrumentation sections of a command can be split between static and dynamic parts of this underlying framework. The block arrows in figure 1 represent the mapping of a Cinnamon program to three different configurations: `A` is static only, a static analyzer with static instrumenter; `B` is hybrid, a static analyzer with a dynamic instrumenter; and `C` is dynamic only, a dynamic analyzer with a dynamic instrumenter. Compilation of a Cinnamon program into framework-specific code that realizes the intended profiling or monitoring technique is discussed in more detail in section IV.

Figure 2 shows the higher-level structure of a program in pseudo-code. This example has a single command that operates on instructions in the binary filtered through the use of a constraint on the command-block declaration. Inside the command, two actions are defined, which are to be executed *before* and *after* the instruction executes. User-defined variables can occur at a global level or within any inner block, and do not escape out of their defining scope. Analysis code is written within the command but not within any action.

B. Language Specification

1) Commands: A profiling or monitoring technique is implemented in Cinnamon using a set of commands. As described earlier, each command specifies the type of a control-flow element (CFE) from the under-observation binary, along with any constraints on its selection, and contains the analysis and instrumentation code to be run on the instances of the selected CFE.

2) Control-Flow Abstractions: Recovery of and access to control-flow structures, such as basic blocks and functions, is a pre-requisite for any static or dynamic analysis on the binary and also one of the most challenging tasks. In Cinnamon, we abstract away the process of control-flow recovery and define special types that represent the different control-flow elements available, including modules, functions, loops, basic blocks and instructions.

A module (`module`) type represents the highest level of the control-flow structure of the binary and comprises of all the functions, loops, basic blocks, and instructions present within a single binary object. The binary executable is a module itself, and any shared libraries that it links to are presented as separate modules. A function (`func`) is a second-level CFE

that further contains any loops, basic blocks and instructions that belong to that function. Loops (`loop`) contain basic blocks and instructions, and are recognised by building the control-flow graph of a function, performing dominance analysis, and identifying back edges. A basic block (`basicblock`) is a sequence of instructions where each instruction has only one predecessor (apart from the first) and only one successor (apart from the last). Finally, an instruction (`inst`), is the lowest-level control-flow element, representing machine instructions from the target framework's intermediate representation. A command can be written to run on any type of element and may contain further commands that run on lower-level elements within the scope of the enclosing command.

3) Actions: An action in Cinnamon is the code to be instrumented at a specified location in the binary. Actions are expressed as a sequence of C-style statements and can contain `for` loops, `if-then-else` conditionals, function calls, assignments, and arithmetic and logic statements as well as I/O operations to standard input/output and files.

4) Trigger Points: We define instrumentation locations, called *trigger points*, related to different CFEs, where profiling or monitoring code can be inserted. For basic blocks, functions, and loops, these are on `entry` and `exit` to the CFE; for instructions they are `before` and `after`. There are two special blocks, `init` and `exit` blocks, which correspond to the locations where any initialisation and finalisation code (i.e., code to be executed before or after any instructions belonging to an application) can be added.

5) Analysis and Inspection Code: Before instrumenting actions for a specific CFE, a programmer may need to define or compute some data, to be used later in the action block, by performing some analysis. In Cinnamon, this analysis code can be placed before an action block and can be written as C-style statements.

6) Constraints: Programmer-specified constraints on command blocks serve as a filtering criteria to avoid running the analysis and instrumentation code where not needed, thus reducing overhead. Based on the target framework and validity of the requested information for filtering, these constraints can be evaluated either statically or dynamically. The compiler backend can check whether the information needed to evaluate a constraint is available and valid at the required program point for the target framework, and throw an error if not.

7) Ordering of Commands and Actions: If multiple commands are present in a program, their code will be mapped (but not necessarily executed) in the same order in the underlying framework. Similarly, if multiple actions are listed for a CFE at the same trigger point (for example, before a certain instruction), they will be instrumented in the same order they occur in the Cinnamon program.

When writing Cinnamon programs, developers need to pay attention to when the analysis and instrumentation of actions is *performed* and when the instrumented actions are actually *executed*. The analysis and instrumentation stage is when the Cinnamon-compiler generated code goes over different CFEs, performs computation or analysis, and instruments the code according to the action, init and exit blocks. Only the code outside action, init and exit blocks is actually executed at this stage. The execution stage is when the newly updated under-observation binary is actually run, along with the instrumented code of action, init and exit blocks.

Different commands may communicate with each other only through global variables or files (via I/O). In addition, if a command produces some data that is later required by another command during the analysis and instrumentation stage, then these commands will need to be listed in the order of producer then consumer.

8) Types and Opcodes: Cinnamon supports a number of primitive and composite types, as well as defining certain special types suitable for binary instrumentation. The primitive types include integers, chars, and booleans, whereas composite types include dictionary or map structures and static and dynamic arrays (or vectors). A special `addr` type can be used to hold the pointer-sized address of a memory location or the target address of a control-transfer instruction.

A number of *storage abstractions* are defined that represent whether the operand of an instruction is a memory location (`mem`), a register (`reg`), or an immediate value (`const`). The keyword `IsType` accesses a compiler builtin that can be used to check the storage type of an operand against one of these abstractions. Cinnamon also defines a list of special keywords that represent instruction opcodes, such as `Call`, `Load`, and `Branch`. The present list of primitive and composite types and opcodes is not fixed and can be easily extended for a new architecture (and ISA).

9) Attributes of Control-Flow Elements: In order to perform profiling and monitoring, a programmer may need to acquire different attributes of a CFE, such as the target address of a call instruction, operand address of a memory instruction, or identifier (ID) of a loop. Accessing the values of these attributes in the target framework is not always trivial and may require writing complex low-level code. Cinnamon simplifies the access to these attributes through the dot (.) operator, in the same way as accessing fields of a `struct` in C.

C. Grammar

Figure 3 shows the partial grammar of Cinnamon in EBNF form. The root of the syntax tree is the Cinnamon *program*, which is composed of optional global variable declarations, a set of *command blocks*, and initialisation and finalisation blocks. We have omitted several productions rules as they can be defined similarly as in standard programming languages, such as C, e.g., ⟨id⟩ for C-style identifiers, ⟨str_const⟩ for string literals, ⟨init_expr⟩ for initializer lists.

IV. Cinnamon Implementation

The workflow in figure 4 shows compilation and translation of Cinnamon code to build a stand-alone profiling or monitoring tool, and is composed of two stages.

The first stage involves the Cinnamon compiler, which consists of a front-end for a Cinnamon program and a back-end

⟨prog⟩ ⊨ ⟨decl⟩* (**init {** ⟨stmt⟩* **}**)? ⟨cmd⟩* (**exit {** ⟨stmt⟩* **}**)?
⟨cmd⟩ ⊨ ⟨e_type⟩ ⟨id⟩ ⟨cst⟩? **{** (⟨cmd⟩ | ⟨stmt⟩ | ⟨act⟩)* **}**
⟨cst⟩ ⊨ **where (**⟨expr⟩**)**
⟨act⟩ ⊨ ⟨trigger_point⟩ ⟨id⟩ ⟨cst⟩? **{** ⟨stmt⟩* **}**
⟨stmt⟩ ⊨ ⟨expr⟩**;** | ⟨decl⟩**;** | ⟨lvalue⟩ **=** ⟨expr⟩**;**
| **if (**⟨expr⟩**) {** ⟨stmt⟩* **}** (**else {** ⟨stmt⟩* **}**)?
| **for (**⟨decl⟩?**;** ⟨expr⟩?**;** ⟨stmt⟩?**) {** ⟨stmt⟩* **}**
⟨expr⟩ ⊨ ⟨lvalue⟩ **IsType** ⟨op_type⟩ | ⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩ | ⟨lvalue⟩ | ⟨rvalue⟩
⟨decl⟩ ⊨ ⟨type-spec⟩ ⟨id⟩ (**=** ⟨init_expr⟩)?
⟨lvalue⟩ ⊨ ⟨id⟩ | ⟨lvalue⟩**[**⟨expr⟩**]** | ⟨lvalue⟩**.**⟨id⟩
⟨rvalue⟩ ⊨ ⟨id⟩**(**⟨args⟩?**)** | ⟨str_const⟩ | ⟨bool_const⟩ | ⟨num⟩ | ⟨opcode⟩ | ⟨null_ptr⟩
⟨args⟩ ⊨ ⟨expr⟩ | ⟨expr⟩**,** ⟨args⟩
⟨e_type⟩ ⊨ **inst** | **basicblock** | **func** | **loop** | **module**
⟨opcode⟩ ⊨ **Call** | **Mov** | **Load** | **Store** | **Branch** | **Return** | **Add** | **Sub** | **Mul** | **Div** | **GetPtr**
⟨op_type⟩ ⊨ **mem** | **reg** | **const**
⟨trigger_point⟩ ⊨ **before** | **after** | **entry** | **exit** | **iter**

Figure 3: The partial grammar of Cinnamon.

or code generator for the target binary framework. The front-end performs lexing and parsing of the code and generates an abstract syntax tree (AST). The code generator processes each AST node and emits C/C++ code that includes target-framework-specific analysis passes, handler passes, which perform instrumentation based on the constraints, and an interface between analysis and handler passes. The front-end of the compiler is independent of the target binary framework and does not need to be changed if the Cinnamon compiler is ported to a different framework.

The second stage of the workflow involves plugging the code generated by the Cinnamon compiler into appropriate places in the target framework through the use of template files, containing boilerplate code, and pattern matching. The combined code is then compiled using a standard compiler (e.g., GCC and LLVM) to build the final tool.

A. Utility Libraries

Section III-B describes how attributes of a CFE can be accessed in Cinnamon using a dot operator, removing the need to write low-level, complex code to access them. To achieve this we provide a number of *get* or accessor-like utility routines that encapsulate the low-level code for each target framework to extract the desired attributes and return the values to the programmer. Since a CFE attribute then becomes just an identifier in Cinnamon, it is simple for a programmer to modify the compiler to extend the attribute list by implementing each new accessor routine and thus create new attributes.

B. Interface between Static and Dynamic Contexts

The attributes of a control-flow element may belong to either or both of its static and dynamic contexts. For example, the IDs assigned to loops or functions statically, at the time of control-flow reconstruction, are not part of their dynamic context. However, during its execution, the instrumented code may need to know the ID of a loop in order to associate profiling data with it. Similarly, the analysis code may generate data that needs to be consumed by the instrumentation code. For such cases, any static attributes of a CFE, or data generated by the analysis code, are passed to the instrumentation code in a framework-dependent manner (for example, by encoding them as arguments to call-back functions).

C. Integration with Binary Frameworks

We target Cinnamon to three different binary analysis and instrumentation frameworks: Janus, Dyninst and Pin.

1) Integration with Janus: Janus is composed of a static analyzer and a dynamic instrumenter. The static part includes core libraries that construct the control-flow graph (CFG) of a binary and provide an interface to analyse instructions, basic blocks, functions, loops, and their attributes. Instructions or basic blocks can be annotated with hints, called *rewrite rules*, which are recorded separately and encode information about a corresponding dynamic handler and any data to be passed along. The dynamic part of Janus is based on DynamoRIO. It dynamically translates the binary one basic block at a time and allows inspection or modification of the constituent instructions according to the rewrite rules before the block is executed for the first time.

When targeting a Cinnamon program to Janus, the code within a command is split between its static and dynamic parts. The Cinnamon compiler emits code for the static analyzer that traverses the list of CFEs based on the constraints specified by the command and executes any analysis code. The compiler

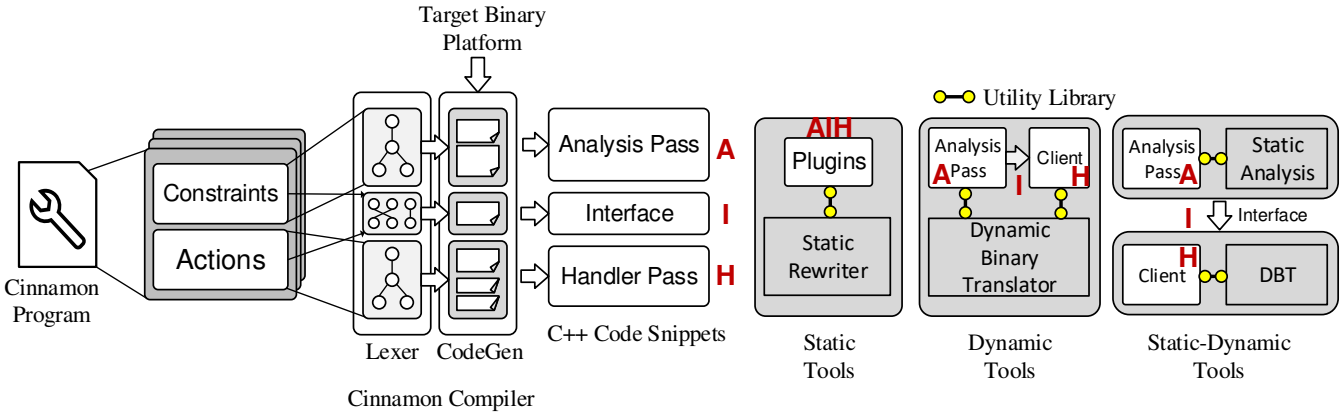

Figure 4: Cinnamon workflow.

adds boilerplate code for the static analyzer that encodes actions and their trigger points in separate rewrite rules. Any data produced by the static analyzer that is later consumed by an action is also recorded in the rewrite rule.

The code emitted for each action is encapsulated in a separate callback function. The Cinnamon compiler further generates boilerplate code for the dynamic part, where instructions within each basic block are checked for any associated rewrite rules. If found, the rewrite rule is decoded to extract the trigger point and the corresponding callback function for the action. The callback function is then instrumented at the specified location using DynamoRIO's *clean call* facility. Any data produced by the static analyzer is passed as one or more arguments to the callback function.

2) Integration with Dyninst: Dyninst allows both static and dynamic instrumentation of a binary. It builds the CFG of a target binary through static analysis and offers control-flow abstractions, such as functions, loops, and basic blocks, to the programmer, as well as handles to different instrumentation locations, such as the entry and exit of a loop. It uses a trampoline-based approach to add instrumentation code, called a *snippet*, to the binary.

We use Dyninst in its static analysis and rewriting mode and implement a back-end for the Cinnamon compiler to generate the code accordingly. Similarly to Janus, code is emitted for each command to iterate over the list of control-flow elements and execute analysis code. Actions are also encapsulated similarly in callback functions and the Cinnamon compiler emits code that uses Dyninst APIs to determine the location of the specified instrumentation point in the binary. The action code is instrumented as snippets and any static analysis data can be directly passed to callback functions as one or more arguments.

3) Integration with Pin: Pin performs dynamic binary instrumentation using a just-in-time compilation approach, where code is instrumented on-the-fly just before it is executed. There are four main instrumentation modes in Pin that allow inspection and instrumentation of a binary at different granularity levels (i.e., instruction, trace, routine and image). Routine and image modes work ahead-of-time and need symbolic information; information about loops is not available in Pin.

Similarly to Janus and Dyninst, the Cinnamon compiler encapsulates actions in callback functions. The code to insert calls to these functions in Pin is enclosed inside instrumentation callback routines that operate in one of the four instrumentation modes based on the granularity specified by the programmer. As such, instrumentation code can be inserted before or after an instruction and at the entry and exit of a function or basic block. All necessary boilerplate code and conversion of data to be passed as arguments to callback functions is generated by the Cinnamon compiler.

D. Summary

We have presented Cinnamon, a domain-specific language for binary profiling and monitoring, with a compiler that targets three instrumentation frameworks. We now describe use cases for our language, before evaluating its usefulness as an abstraction tool.

V. Case Studies

We present a number of profiling and monitoring codes as case studies to demonstrate the breadth of applications that can be built with Cinnamon.

A. Profiling Tools

1) Instruction Counting: Instruction counting is a simple and common profiling application. We demonstrate two different ways to implement it in Cinnamon.

The code listed in figure 5a is a basic way to count the number of load instructions in a program. The command on lines 2–6 selects all load instructions and applies an action to increment a global counter `inst_count` before each of those instructions executes. The final value of the counter is printed to standard output on program exit (lines 7–9).

In contrast, the code shown in figure 5b implements a lower overhead version of instruction counting. It keeps a local counter of load instructions for each basic block (lines 3–6) and defines an action to add the local counter to a global counter before each block executes (line 7–9). This results in

```
1 uint64 inst_count = 0;
2 inst I where (I.opcode == Load) {
3   before I {
4     inst_count = inst_count + 1;
5   }
6 }
7 exit {
8   print(inst_count);
9 }
```

(a) Basic

```
1  uint64 inst_count = 0;
2  basicblock B {
3    uint64 local_inst_count = 0;
4    inst I where (I.opcode == Load) {
5      local_inst_count = local_inst_count + 1;
6    }
7    before B where (local_inst_count > 0) {
8      inst_count = inst_count + local_inst_count;
9    }
10 }
11 exit {
12   print(inst_count);
13 }
```

(b) Lower overhead, demonstrating nested commands

Figure 5: Two implementations of instruction counting.

action code being instrumented and executed once per basic block instead of once per load instruction and the overhead is thus reduced compared to the first code. The nested command on lines 4–6 is within the scope of the basic block defined by the outer command and contains no action code, hence the statement on line 5 is part of the analysis phase only. Thus, the computation of `local_inst_count` is complete before the action on lines 7–9 is instrumented for basic block `B`.

The Cinnamon compiler encapsulates the code within action blocks at figure 5a, line 4 and figure 5b, line 8 in callback functions. For Janus, the compiler emits code to dynamically insert clean calls to these functions whereas the rest of the code is mapped to the static analyzer. Both the analysis and instrumentation of actions is done statically for Dyninst and dynamically for Pin.

2) Loop-Coverage Profiling: Loop-coverage profiling is an important tool that can be used to identify hot loops, which are potential targets for optimization, such as parallelization. Figure 6 shows the code for a loop-coverage profiler in Cinnamon. At the entry block of each loop, during its execution, the loop is marked as an active or living loop (lines 10–12) and set back to inactive at the exit block of the loop (lines 13–15). Next, at the entry to each basic block, a global counter of executed blocks is incremented and local counter of each active loop is also incremented (lines 23–31) to record that this basic block has been executed. At program exit, the coverage of each loop is calculated as a percentage of the total number of basic blocks executed in the program (lines 31–36).

B. Monitoring Tools

1) Use-After-Free Vulnerability Monitoring: Use-after-free vulnerabilities occur when a programmer frees memory (in a language without garbage collection) but holds on to a pointer

```
1  uint64 global_bbs_exec = 0;
2  int next_global_id = 0;
3  dict<int, int> loop_bbs_exec;
4  dict<int, int> living_loop;
5  int num_loops = 0;
6
7  loop L {
8    int loop_id = next_global_id;
9    next_global_id = next_global_id + 1;
10   entry L {
11     living_loop[loop_id] = 1;                  // loop becomes alive
12   }
13   exit L {
14     living_loop[loop_id] = 0;                 // loop no longer alive
15   }
16 }
17 module M {
18   num_loops = num_loops + M.numLoops;
19   basicblock B {
20     entry B {
21       global_bbs_exec = global_bbs_exec + 1;
22       for (int i=0; i<num_loops; i=i+1) {
23         if (living_loop[i] == 1) {
24           // update loop count for all alive loops
25           loop_bbs_exec[i] = loop_bbs_exec[i] + 1;
26         }
27       }
28     }
29   }
30 }
31 exit {
32   for (int i=0; i<num_loops; i=i+1) {
33     uint64 count = (loop_bbs_exec[i] / global_bbs_exec) * 100;
34     print(count);
35   }
36 }
```

Figure 6: Loop-coverage profiler.

```
1  dict<addr,int> freed;
2  dict<addr,addr> base_table;
3  int size;
4
5  inst I where (I.opcode == Call && I.trgname == "malloc") {
6    before I {
7      size = I.arg1;                        // size of malloc allocation
8    }
9    after I {
10     addr base_addr = I.rtnval;          // base address of allocation
11     for (addr i=base_addr; i<base_addr+size; i=i+1) {
12       base_table[i] = base_addr;
13     }
14     freed[base_addr] = 0;
15   }
16 }
17 inst I where (I.opcode == Call && I.trgname == "free") {
18   before I {
19     addr ptr_addr = I.arg1;                    // address to be freed
20     freed[ptr_addr] = 1;
21   }
22 }
23 inst I where (I.opcode == Load || I.opcode == Store) {
24   before I {
25     addr acc_addr = I.memaddr;          // address being read/written
26     addr base_addr;
27     if (base_table[acc_addr] != NULL ) {
28       base_addr = base_table[acc_addr];
29       if (freed[base_addr] == 1) {
30         print("ERROR:_use_after_free_access");
31       }
32     }
33   }
34 }
```

Figure 7: Use-after-free vulnerability monitoring.

to that memory that is later reused. They can be exploited by attackers by reallocating the memory to themselves and leveraging the reuse to take over the program.

Figure 7 shows code written in Cinnamon that monitors and intercepts the occurrence of use-after-free accesses [47]. The command on lines 5–18 selects call instructions to the `malloc` function. For each such instruction, it records the size, which is the first argument of the malloc call, and the base address of the allocation, which is the value returned by the malloc call (line 6–15), and marks as allocated addresses in the range `<base_addr, base_addr+size>`. The second command (lines 17–22) selects all call instructions to `free` and defines the action to mark the input addresses as freed.

The last command (lines 23–35) selects all load and store instructions and checks whether they represent a use-after-free access, i.e., the address being read or written having initially been allocated by a malloc call and later freed.

2) Control-Flow Integrity—Shadow Stack: Shadow stack is a defense mechanism used to implement control-flow integrity on backward edges (i.e., function returns) [48]. An attacker can divert the execution of the program to malicious code by changing the contents of the function stack, including the return address, through a buffer-overflow attack.

Figure 8 shows Cinnamon code that implements the stricter form of backward CFI, allowing a callee to return only to its most recent caller. For each call instruction, we record the address it should return to on a shadow stack (lines 4–10). Next, we specify code for all return instructions that checks the return address *before* each is executed and ensures that it matches the address on top of shadow stack (lines 11–19).

3) Control-Flow Integrity—Forward Edge: Control-flow integrity is also used to protect forward edges, i.e., function calls and branches. Direct function calls and branches are not an issue, since their target is part of the code memory and hence not writeable. However, the targets of indirect calls and branches are calculated from the contents of data memory, which can be written to and corrupted.

Figure 9 shows our implementation of forward CFI, which allows any valid function to be the target of a call. We first record the start address of each function (lines 4–6) in a file. These values are read by initialization code in a vector `vtable` (lines 14–17). The command on lines 7–13 specifies an action *before* each call instruction that checks whether the target address is one of the valid function addresses (i.e., found in the vector`vtable`).

```
1  dict<int,addr> sstack;
2  int top = 0;
3
4  inst I where (I.opcode == Call) {
5    before I {
6      addr fall_addr = I.nextaddr;          // return address of call
7      sstack[top] = fall_addr;
8      top = top + 1;
9    }
10 }
11 inst I where (I.opcode == Return) {
12   before I {
13     if (top > 0 && sstack[top-1] == I.trgaddr) {
14       top = top - 1;
15     } else{
16       print("ERROR");
17     }
18   }
19 }
```

Figure 8: Shadow stack.

```
1  vector<addr> vtable;
2  file outfile("fAddr.txt");
3
4  func F {
5    writeToFile(outfile, F.startAddr);
6  }
7  inst I where (I.opcode == Call) {
8    before I {
9      if (!vtable.has(I.trgaddr)) {
10       print("ERROR");
11     }
12   }
13 }
14 init {
15   for(line l = outfile.getline(); l!=NULL; )
16       vtable.add(l);
17 }
```

Figure 9: Forward control-flow integrity.

VI. Discussion and Evaluation

Cinnamon's value can be considered based on a number of factors, such as whether it is expressive enough to implement different binary profiling techniques, the length of code to write a certain technique, its portability to different frameworks and platforms, as well as its performance. We discuss these factors along with any limitations.

A. Expressiveness

The abstractions for control-flow elements and instrumentation locations defined by Cinnamon allow flexibility to insert code at different granularities and enable the implementation of a wide range of profiling and monitoring techniques. Instruction counting and loop-coverage profiling use cases demonstrate the flexibility of Cinnamon to perform profiling at both fine- and coarse-grained levels.

The field dot operator notation, such as `I.opcode`, facilitates a uniform interface to access different attributes of a CFE in both static and dynamic domains. Any new fields or attributes related to a CFE can be supported by implementing the corresponding accessor function.

The analysis code and actions can be represented in C-style along with a number of supported data structures and I/O operations. This makes the analysis and instrumentation code very expressive and enables complex profiling and monitoring techniques to be applied.

B. Code Length and Complexity

In order to build a custom profiling or monitoring tool using existing binary frameworks, programmers have to write framework-specific code for set-up or tear-down, insertion of callbacks, and access to attributes. In Cinnamon, we shift the burden of adding this code from the developer to Cinnamon's back-end. This reduces the number of lines of code that a programmer has to write, as well as the complexity of the

TABLE I: COMPARISON OF CODE LENGTHS.

Use case	Cinnamon	Dyninst	Janus	Pin
Inst count	10	215	90	23
Loop coverage	40	229	150	-
Use-after-free	39	260	193	57
Shadow stack	20	196	136	38
Forward CFI	17	207	103	50

code, and helps the programmer to focus on implementing their specific profiling or monitoring technique and not worry about lower-level framework details. Any boilerplate code needs to be written only once by the back-end developer.

Table I shows a comparison of the number of lines of code written for the use cases from section V when implemented in Cinnamon versus directly in Dyninst, Janus and Pin. Dyninst, in particular, requires verbose code, but Cinnamon still needs almost an order of magnitude less code than Janus. Note also that writing a Cinnamon program just once allows it to be targeted to any of these three frameworks without modification, instead of having to develop separate versions for each tool, as explained next. The only exception here is the loop coverage example which could not be translated to Pin in its original form as Pin does not have a notion of loops. However, integrating loop detection techniques [49] in Pin could make it transparent to the programmer.

C. Portability

Cinnamon captures the abstractions, types, and code patterns that are specific to binary instrumentation but are independent of the underlying framework being used. Hence, the programmer can port or re-target their code to a different framework just by using its respective back-end and without having to rewrite the code.

Figure 10 shows a comparison of code in Dyninst and Janus to access the return value of a function call, using x86 calling conventions, and pass it to an instrumented callback function `on_call`. Similarly, figure 11 shows the code to access a store's memory address and pass it to an instrumented callback function `on_store`. In Cinnamon, the value of these parameters can be accessed simply by using `I.rtnval` and `I.dstaddr` notations respectively; the corresponding code is hidden behind and enclosed in utility library routines.

Figure 12 presents a comparison of the number of instructions in SPEC CPU 2017 benchmarks as reported by Cinnamon's instruction-counting program, targeted to the frameworks we have described. As can be seen, instruction counts are consistent between these back-ends, despite using exactly the same Cinnamon program. Pin, however, reports a significantly higher count for *omnetpp*, *exchange*, *bwaves* and *fotoni3d* as it performs dynamic instrumentation and hence is able to discover more instructions than Janus and Dyninst (i.e., it counts instructions within dynamically linked libraries). Unfortunately, several benchmarks would not work correctly with Dyninst, either when targeted by Cinnamon's compiler, or when writing code directly for that framework due to incomplete or imprecise control-flow recovery.

```
/*----- callback function ----*/
void on_call(uint64_t rtn_val){
  printf("return value: %ld\n", rtn_val);
}
/*- code to access return value & add instrumentation -*/
BPatch_Vector<BPatch_snippet*> instArgs;
BPatch_retExpr rtn_val;
instArgs.push_back(&rtn_val);
BPatch_funcCallExpr instrFuncExpr(*on_call, instArgs);
BPatchSnippetHandle *handle = app->insertSnippet(instrFuncExpr,
    *instr, BPatch_callAfter);
```

(a) Dyninst

```
/*----- callback function ----*/
void on_call(uint64_t rtn_val){
  printf("return value: %ld\n", rtn_val);
}
/*- code to access return value & add instrumentation -*/
dr_insert_clean_call(drcontext, bb, instr, (void*)on_call,
    false, opnd_create_reg(DR_REG_RAX));
```

(b) Janus

Figure 10: Code to access the return value of a call instruction `instr` in Dyninst and Janus.

```
/*----- callback function ----*/
void on_store(intptr_t dst_addr){
  printf("dest addr: %p\n", dst_addr);
}
/*- access destination memory address & add instrumentation -*/
BPatch_effectiveAddressExpr dst_addr;
BPatch_Vector< BPatch_snippet*> instArgs;
instArgs.push_back(&dst_addr);
BPatch_funcCallExpr instrFuncExpr(*on_store, instArgs);
BPatchSnippetHandle *handle = app->insertSnippet(instrFuncExpr,
    *instr, BPatch_callBefore);
```

(a) Dyninst

```
/*----- callback function ----*/
void on_store(app_pc st_instr){
    app_pc dst_addr;
    instr_t decoded_instr;
    int num_dsts;
    dr_mcontext_t mc = {sizeof(mc), DR_MC_ALL};
    void *drcontext = dr_get_current_drcontext();
    dr_get_mcontext(drcontext, &mc);

    instr_init(drcontext,&decoded_instr);
    if(decode(drcontext, st_instr, &decoded_instr) != NULL){
        num_dsts = instr_num_dsts(&decoded_instr);
        for(int i=0; i<num_dsts; i++){
            opnd_t mem_opnd = instr_get_dst(&decoded_instr,i);
            dst_addr = opnd_compute_address(mem_opnd, &mc);
        }
    }
    printf("dest addr: %p\n", dst_addr);
}
/*- add instrumentation -*/
app_pc src_instr = instr_get_app_pc(instr);
dr_insert_clean_call(drcontext, bb, instr, (void*)on_store,
    false, 1, OPND_CREATE_INTPTR(src_instr));
```

(b) Janus

Figure 11: Code to access the memory address being written by a store instruction `instr` in Dyninst and Janus.

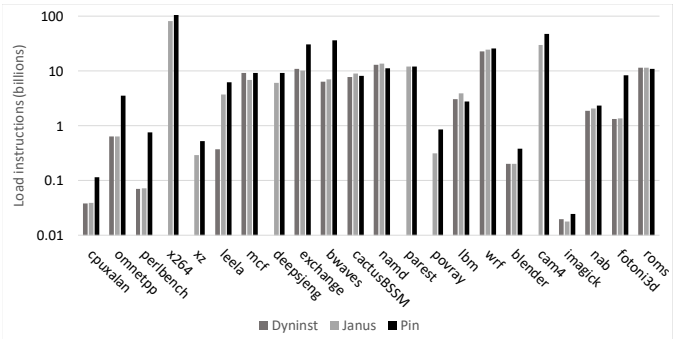

Figure 12: Number of load instructions in SPEC CPU 2017 benchmarks (*test* input) as reported by Cinnamon back-ends.

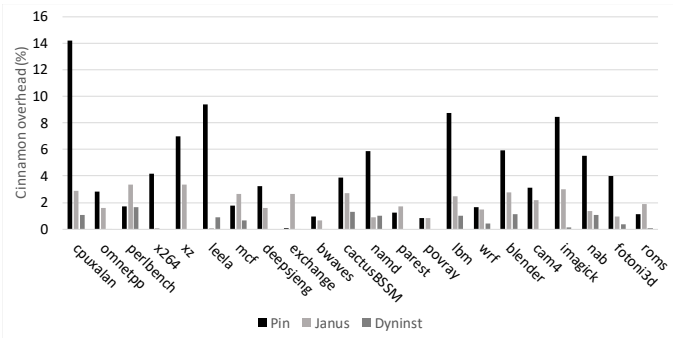

Figure 13: Cinnamon overhead (%) for instruction counting (figure 5b) across SPEC CPU 2017 benchmarks compared to implementation of the same profiling tool natively in the underlying framework.

D. Performance

The performance of the final under-observation binary is dependent on the instrumentation code that is added by the developer in their Cinnamon program and not as a result of overheads that Cinnamon's compiler introduces. However, we encapsulate the action code in callback routines and use clean calls in DynamoRIO and Pin, and function-call snippets in Dyninst to execute them. Due to context switches, callback routines may have a higher overhead than inserting the action code directly into the binary using low-level instructions. However, the underlying binary framework may still inline the clean call (as DynamoRIO does) if the code within the callback function is simple enough (i.e., it does not involve calls to other functions).

To demonstrate the overheads we experienced for the tools developed in this paper, we compared execution times of profiling tools we wrote ourselves directly in each framework and the tools developed with Cinnamon across SPEC CPU 2017 benchmarks. Figure 13 shows the overhead for the instruction counting tool from figure 5b, one of the simplest realistic tools that could be developed and therefore not reliant on the skill of the programmer to write (either natively or in Cinnamon). Cinnamon's overheads are highest for Pin (average 4.75%), then Janus (average 1.88%) and finally Dyninst (average 0.67%), although the latter does not work for all benchmarks, as previously described. Overheads are negligible for many benchmarks and all under 4% for Janus and Dyninst. Considering Pin only, with the highest overheads, other tools are similar—use-after-free monitoring (figure 7) has 0.52% overhead on average and 1.78% maximum, whereas forward CFI (figure 9) has 3.06% overhead on average and 11% maximum.

E. Limitations

The control-flow abstractions provided by Cinnamon essentially rely upon the support of corresponding features provided by underlying target frameworks. For example, Pin does not have the notion of loops thus the granularity levels at which code can be inserted will be limited. Similarly, access to various attributes of a CFE depend on whether the corresponding accessor routines are implemented in utility libraries. In other words, some dynamic attributes may not be available for purely static techniques and vice versa. Currently, we only provide accessor routines for the attributes, which means they cannot be modified. This is sufficient for our purposes as Cinnamon programs are meant for performing only passive monitoring. In the future, Cinnamon could be extended with modifier routines to allow more transformations.

Further, Cinnamon does not support interrupt routines, such as those based on timers. Therefore, timer-based sampling techniques, which may have lower runtime overhead than sampling every instruction or basic block, cannot be implemented currently. Cinnamon does not currently contain thread-related primitives, but can instrument multi-threaded applications if the analysis does not require synchronization.

VII. Related Work

In recent years, there has been a significant interest in the development of domain-specific languages for analysis and optimization at source code [50], [51], compiler [52]–[55], and binary [56]–[59] levels.

CanDL [52] is a constraint-based DSL to write compiler-analysis code that operates over LLVM's IR. Users provide analysis specifications through constraints that are translated down to an LLVM analysis pass; transformations must be written separately in C++. CanDL can be used for both peephole and more complex analysis algorithms with fewer lines of code when compared to LLVM. However, the specification for CanDL comprises of a large number of varied and non-uniform constructs and constraints that users need to familiarize themselves with before being able to write any analysis code. In contrast, Alive [53] allows peephole optimizations through specification of pattern or code sequences in the LLVM IR that can be replaced with more optimized sequences. The ER-ESI [60] project on reverse engineering of binaries introduces a RISC-like intermediate representation langauge called ELIR to facilitate binary analysis, along with a domain-specific meta langauge [61] to write analysis routines that operate on this representation. REIL [62] is another intermediate-representation language developed to allow writing analysis routines in a platform-independent manner and has been used as the base representation in other binary-analysis frameworks [63].

MDL [59] is a DSL that facilitates collection of performance data through dynamic binary instrumentation at function entry, exit, and call points but is limited to collecting performance metrics through counters and timers. In contrast, DiSL [57] is a bytecode instrumentation language embedded in Java to support dynamic program analysis by inserting code snippets in the form of Java classes. Mussler et al. [58] propose an instrumentation DSL built on top of Dyninst that uses the static structural information to filter the instrumentation locations in the code and reduce the measurement overhead. The instrumentation code snippet and filters can be configured through adapter specifications, an XML file, and user filters, but it is not clear how these interact. Moreover, the filtering criteria have been defined only in the context of functions. It is not clear if and how these criteria can be extended or customized and whether they can be applied to other code structures, such as loops and call sites.

There has also been some work in performance-oriented languages that focuses on specifying algorithmic skeletons or code patterns that can be used to find their most efficient and optimized implementations. Lift [64] specifies high-level algorithmic expressions that can be translated down to OpenCL implementations for different platforms. The Idiom Description Language (IDL) [54] describes idioms or patterns in programs so that legacy sequential codes can be optimized and targeted to use different optimized APIs and DSLs for linear algebra libraries, such as BLAS and stencil computation. Halide [65] is a DSL for writing image-processing pipelines that separates the concern of specifying computation and how the computation is performed (i.e., through tiling, vectorization, or parallelization).

VIII. Conclusion and Future Work

Building binary profiling and monitoring tools from scratch or using existing binary frameworks can be a very tedious and time-consuming task. Cinnamon enables a flexible and framework-independent way of building such tools that allows the developer to focus on implementing a specific profiling and monitoring approach. We demonstrate, through different use-cases, that Cinnamon can be used to express a variety of profiling and monitoring techniques and can be ported across frameworks without having to re-write the code.

Cinnamon allows analysis or instrumentation to be performed on a binary without altering the original application code. Future work will investigage how Cinnamon can be extended to support optimization, ranging from peephole transformations to parallelization.

References

[1] J. Kinder, "Static analysis of x86 executables," Ph.D. dissertation, Technische Universität Darmstadt, 2010.

[2] X. Meng and B. P. Miller, "Binary code is not easy," in *ISSTA*, 2016.

[3] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in {*USENIX*} *Security Symposium*, 2015.

[4] "Hex Rays, IDA Pro," https://www.hex-rays.com/products/ida/.

[5] S. B. Yadavalli and A. Smith, "Raising binaries to LLVM IR with MCTOLL," in *LCTES*, 2019.

[6] A. Dinaburg and A. Ruef, "Mcsema: Static translation of x86 instructions to llvm," in *ReCon*, 2014.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Computer Aided Verification*, 2011.

[8] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, 2005.

[9] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[10] 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 *PLDI*, 2005.

[11] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *CGO*, 2003.

[12] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, 2008.

[13] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *PASTE*, 2011.

[14] A. Eustace and A. Srivastava, "ATOM: A flexible interface for building high performance program analysis tools," in *USENIX Technical Conference*, 1995.

[15] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for linux," in *ISPASS*, 2010.

[16] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[17] N. Nethercote, R. Walsh, and J. Fitzhardinge, "Building workload characterization tools with valgrind," in *IISWC*, 2006.

[18] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Security and Privacy*, 2015.

[19] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Security and Privacy*, 2006.

[20] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *USENIX Security Symposium*, 2013.

[21] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure execution via program shepherding," in *USENIX Security Symposium*, 2002.

[22] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *CGO*, 2011.

[23] S. Amarasinghe, "Secure execution environment via program shepherding," in *USENIX Security Symposium*, 2002.

[24] M. Prasad and T.-c. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks." in *USENIX Annual Technical Conference*, 2003.

[25] N. Nethercote and J. Fitzhardinge, "Bounds-checking entire programs without recompiling," in *SPACE*, 2004.

[26] R. Zhou, G. Wort, M. Erdős, and T. M. Jones, "The Janus triad: Exploiting parallelism through dynamic binary modification," in *VEE*, 2019.

[27] B. De Sutter, B. De Bus, and K. De Bosschere, "Link-time binary rewriting techniques for program compaction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 5, 2005.

[28] S. Bansal and A. Aiken, "Binary translation using peephole superoptimizers," in *OSDI*, 2008.

[29] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A practical binary optimizer for data centers and beyond," in *CGO*, 2019.

[30] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *EuroSys*, 2013.

[31] GrammaTech, "https://www.grammatech.com/codesurfer-binaries," 2020.

[32] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Compiler Construction*, 2004.

[33] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting security in COTS software with binary rewriting," in *Future Challenges in Security and Privacy for Academia and Industry*, 2011.

[34] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries," in *NDSS*, 2015.

[35] B. De Sutter, B. De Bus, K. De Bosschere, and S. Debray, "Combining global code and data compaction," in *LCTES*, 2001.

[36] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "Diablo: a reliable, retargetable and extensible link-time rewriting framework," in *International Symposium on Signal Processing and Information Technology*, 2005.

[37] R. Muth, S. Debray, S. Watterson, and K. De Bosschere, "Alto: A link-time optimizer for the Compaq Alpha," *Software: Practice and Experience*, vol. 31, 2001.

[38] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *Security and Privacy*, 2016.

[39] Valgrind Developers, "Callgrind: a call-graph generating cache and branch prediction profiler," 2010.

[40] ——, "Cachegrind: a cache and branch-prediction profiler," 2009.

[41] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *USENIX Annual Technical Conference*, 2005.

[42] C. Wang, S. Hu, H.-s. Kim, S. R. Nair, M. Breternitz, Z. Ying, and Y. Wu, "StarDBT: An efficient multi-platform dynamic binary translation system," in *Advances in Computer Systems Architecture*, L. Choi, Y. Paek, and S. Cho, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 4–15.

[43] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil *et al.*, "Analyzing parallel programs with pin," *IEEE Computer*, vol. 43, no. 3, 2010.

[44] W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," in *ISCC*, 2006.

[45] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *MICRO*, 2006.

[46] J. Marathe, F. Mueller, T. Mohan, S. A. Mckee, B. R. De Supinski, and A. Yoo, "METRIC: Memory tracing via dynamic binary rewriting to identify cache inefficiencies," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 2, 2007.

[47] S. Makarov, "An event-based language for programmable debugging," Ph.D. dissertation, University of Toronto, 2018.

[48] M. Zhang, "Static binary instrumentation with applications to COTS software security," Ph.D. dissertation, Stony Brook University, 2015.

[49] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri, "Loopprof: Dynamic techniques for loop detection and profiling," in *WBIA*, 2006.

[50] T. S. F. X. Teixeira, C. Ancourt, D. Padua, and W. Gropp, "Locus: A system and a language for program optimization," in *CGO*, 2019.

[51] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A domain specific language for source code analysis and manipulation," *International Working Conference on Source Code Analysis and Manipulation*, 2009.

[52] P. Ginsbach, L. Crawford, and M. F. P. O'Boyle, "CAnDL: A domain specific language for compiler analysis," in *CC*, 2018.

[53] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably correct peephole optimizations with Alive," in *PLDI*, 2015.

[54] P. Ginsbach, T. Remmelg, M. Steuwer, B. Bodin, C. Dubach, and M. F. P. O'Boyle, "Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach," in *ASPLOS*, 2018.

[55] D. L. Whitfield and M. L. Soffa, "An approach for exploring code improving transformations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 6, 1997.

[56] A. S. Charif-Rubial, D. Barthou, C. Valensi, S. Shende, A. Malony, and W. Jalby, "MIL: A language to build program analysis tools through static binary instrumentation," in *International Conference on High Performance Computing*, 2013.

[57] L. Marek, A. Villazon, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, "DiSL: A domain-specific language for bytecode instrumentation," in *AOSD*, 2012.

[58] J. Mußler, D. Lorenz, and F. Wolf, "Reducing the overhead of direct application instrumentation using prior static analysis," in *Euro-Par*, 2011.

[59] J. K. Hollingsworth, O. Niam, B. P. Miller, Zhichen Xu, M. J. R. Goncalves, and Ling Zheng, "MDL: a language and compiler for dynamic program instrumentation," in *PACT*, 1997.

[60] J. Vanegue, J. Medeiros, E. Bisolfati, A. Desnos, T. Figueredo, T. Garnier, R. Lesniak, J. Palencia, S. Roy, S. Soudan *et al.*, "The ERESI reverse engineering software interface," 2009.

[61] J. Vanegue, "Static binary analysis with a domain specific language," 2008.

[62] T. Dullien and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," 2009.

[63] C. Heitman and I. Arce, "BARF: a multiplatform open source binary analysis and reverse engineering framework," in *Congreso Argentino de Ciencias de la Computación*, 2014.

[64] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code," in *ICFP*, 2015.

[65] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Transactions on Graphics*, vol. 31, no. 4, 2012.